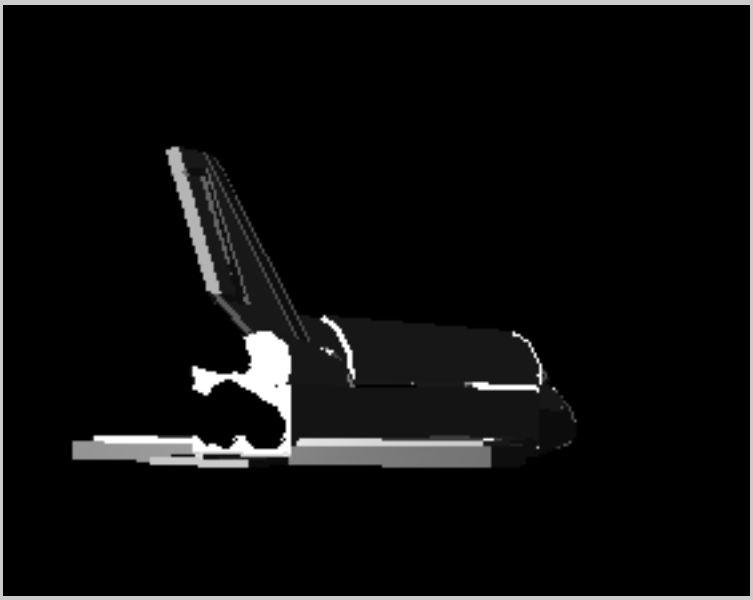
Princeton SATELLITE SYSTEMS

Spacecraft Control Toolbox User's Guide V3.0



This software described in this document is furnished under a license agreement. The software may be used, copied or translated into other languages only under the terms of the license agreement.

Spacecraft Control Toolbox User's Guide (April 1999)

© Copyright 1996-1999 by Princeton Satellite Systems, Inc. All rights reserved.

Any provision of Princeton Satellite System Software to the U.S. Government is with "Restricted Rights" as follows: Use, duplication, or disclosure by the Government is subject to restrictions set forth in subparagraphs (a) through (d) of the Commercial Computer Restricted Rights clause at FAR 52.227-19 when applicable, or in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, and in similar clause in the NASA FAR Supplement. Any provision of Princeton Satellite Systems documentation to the U.S. Government is with Limited Rights. The contractor/manufacturer is Princeton Satellite Systems, Inc., 33 Witherspoon Street, Princeton, New Jersey 08542.

MATLAB is a registered trademark of The MathWorks, Inc.
Macintosh is a registered trademark of Apple Computer, Inc.
Silicon Graphics is a registered trademark os Silicon Graphics
Adobe, Acrobat and FrameMaker are trademarks of Adobe Systems Incorporated.
All other trademarks are the property of their respective holders.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Printing History

June 1999First Printing

Princeton Satellite Systems, Inc.
33 Witherspoon Street
Princeton, New Jersey 08542

Technical Support/Sales/Info:http://www.psatellite.com

Contents

Chapter 1 Installing The Spacecraft Control Toolbox 1-1

Requirements 1-2
About this Tutorial 1-2
Background 1-2
CD-ROM 1-3
Web 1-3
Included Files 1-8
Installation 1-8
Case Sensitivity 1-8
Function Overloading 1-9
Updates 1-9
Technical Support 1-9
Webboard 1-9

Chapter 2 Getting Help 2-1

Demos 2-2
File Help 2-5
Introduction 2-5
The List Pane 2-5
The Example Pane 2-7
Find All Button 2-7
Edit Button 2-8
Search File Names Button 2-8
Search Headers Button 2-9
Search String Edit Box 2-9
Run Example Button 2-10
Save Example Button 2-10
Help Button 2-10
Quit 2-10
Graphical User Interface Help 2-10
MATLAB Help 2-12
Technical Support 2-12
Webboard 2-12

Chapter 3 Using Databases 3-1

The Constant Database 3-2
Merging Constant Databases 3-4

Chapter 4 Basic Functions 4-1

Introduction 4-2
Function Features 4-4
Built-in demos 4-4
Default parameters 4-5
Built-in plotting 4-5
Error checking 4-6
Variable inputs 4-6
Example Functions 4-7
Attitude 4-7
Control 4-8
Coord 4-12
Dynamics 4-12
Environs 4-13
Ephem 4-14
Graphics 4-14
Hardware 4-15
Orbit 4-16
Prop 4-16
Thermal 4-17

Chapter 5 Data Structures 5-1

Data Structures 5-2
Cell Arrays 5-3
Classes 5-4
SCT Classes 5-5

Chapter 6 Coordinate Transformations 6-1

Transformation Matrices 6-2
Quaternions 6-2

Chapter 7 Using the CAD Functions 7-1

Introduction 7-2
Building a Spacecraft Model 7-2
Introduction 7-2
Introduction 7-3
Adding a Component 7-3
Adding a Body 7-9
Linking Bodies 7-10
Displaying a Spacecraft Model 7-10
Interfacing to the Tree Model 7-11
Computing Disturbances 7-12
Importing CAD Models 7-13

Chapter 8 UIElements 15

UIElement 16
EditPM 17
EditScroll 17
CheckBoxList 17
DataStructureList 18
List 18
ListWithButton 19
Closing UIElements 19

Chapter 9 GUI Plug Ins 21

GUIPlugIn 22
Writing Your Own GUI Function 26

Chapter 10 A Worked Design Example 8-1

The MAP Mission 8-2
Control Modes 8-2
Sensing and Actuation 8-3
Control System Design 8-3
Simulation Results 8-3
Using the Spacecraft Control Toolbox 8-4
Summary 8-14

Chapter 11 Simulation 9-1

Dynamics Models 9-3
Wire Model 9-6
Tree Model 9-12
Orbit Simulation 9-16
Introduction 9-16
NORAD Elements 9-16

Chapter 12 Orbit Propagation 19

Orbit Propagation 20
Introduction 20
Limitations 20
Using the Graphical User Interface 21
Overview 22
Panes 22
Setting Up a Batch Script 29
Plugin Functions 30
Building a Plugin Function 34
Reference Frames 38
Coordinate Transformations 38
Finding ECI/ECR coincidence 38
Sun 39
Interfacing to STK 39
Ground Coverage 40

Chapter 13 Budgets 10-1

Pointing Budget 10-2
Fuel Budgets 10-3

Figures, Tables and Listings

Chapter 1 Installing The Spacecraft Control Toolbox 1-1

Figure 1-1 Home page for the PSS website 1-4
Figure 1-2 The upgrades page 1-5
Figure 1-3 Password and user id 1-6
Figure 1-4 Upgrades 1-7
Table 1-1 Archives 1-7
Table 1-2 Files included in the Spacecraft Control Toolbox 1-8
Figure 1-5 Webboard 1-10

Chapter 2 Getting Help 2-1

Figure 2-1 DemoSC 2-2
Figure 2-2 Opening ACSDesigns 2-3
Figure 2-3 Edit Window 2-3
Figure 2-4 Time-to-Go Display 2-4
Figure 2-5 Results of `TGimbal` 2-4
Figure 2-6 The file help GUI 2-5
Figure 2-7 Hierarchical file display 2-6
Figure 2-8 Open directory 2-7
Figure 2-9 Find all 2-8
Figure 2-10 Find “orbit” in the file names 2-9
Figure 2-11 Progress bar 2-9
Figure 2-12 SCHelp GUI 2-10
Figure 2-13 An open directory in SCHelp 2-11
Figure 2-14 Help 2-11
Figure 2-15 Webboard 2-12

Chapter 3 Using Databases 3-1

Figure 3-1 Constant database 3-2
Figure 3-2 Searching for `'mass'` 3-3
Figure 3-3 MergeConstantDB function 3-4

Chapter 4 Basic Functions 4-1

Table 4-1 Toolbox Organization 4-2

Table 4-2 Features in SCT functions 4-4
Figure 4-1 Atmospheric density from `AtmDens2` 4-5
Figure 4-2 Dual Spin Stability 4-7
Figure 4-3 `MagTComp` 4-8
Figure 4-4 LeadLag 4-9
Figure 4-5 Torque transmission 4-10
Figure 4-6 Nichols plot 4-11
Figure 4-7 Bode Plot 4-11
Figure 4-8 A`tmDens1` 4-13
Figure 4-9 `AtmDens2` 4-14
Figure 4-10 A sine wave using Plot2D 4-15
Figure 4-11 Pivot Mechanism 4-15
Figure 4-12 Elliptical orbit 4-16
Figure 4-13 Blowdown curve 4-17
Figure 4-14 Earth Albedo 4-17

Chapter 5 Data Structures 5-1

Table 5-1 Object oriented design terms 5-4
Table 5-2 Classes 5-5
Figure 5-1 Second order response 5-6
Listing 5-1 Example of using the statespace class 5-6

Chapter 6 Coordinate Transformations 6-1

Chapter 7 Using the CAD Functions 7-1

Table 7-1 `Build3DSC` actions 7-2
Figure 7-1 Build3DSC 7-3
Listing 7-1 Adding a component by defining faces and vertices 7-4
Figure 7-2 Geometry of a component 7-5
Table 7-2 Predefined surfaces 7-5
Table 7-3 Geometric primitives 7-6
Table 7-4 Component properties 7-7
Listing 7-2 Example body data structure 7-9
Figure 7-3 Topological tree 7-10
Table 7-5 DrawSC Modes 7-10
Figure 7-4 DrawSC 7-11
Table 7-6 Disturbance 7-12
Figure 7-5 Disturbance display 7-13
Table 7-7 CAD File Importing 7-13

Chapter 8 UIElements 15

Figure 8-1 UIElementDemo 16
Listing 8-1 EditPM 17
Listing 8-2 EditScroll 17
Listing 8-3 CheckBoxList 17
Listing 8-4 DataStructureList 18
Listing 8-5 List 18
Listing 8-6 ListWithButton 19

Chapter 9 GUI Plug Ins 21

Figure 9-1 GUIPlugIn on starting. 22
Figure 9-2 GUIPlugIn after changing inputs 23
Figure 9-3 GUIPlugIn with the earth as center 23
Figure 9-4 Selecting planets in the elements plugin 24
Figure 9-5 GUIPlugIn at the end of the simulation 25
Figure 9-6 GUIPlugIn 3D Spacecraft Display 25
Listing 9-1 GUIPlugin 26
Listing 9-2 CreateGUI 28
Listing 9-3 Initializing the simulation 29
Listing 9-4 The simulation loop. 30
Listing 9-5 Plotting 30
Listing 9-6 Update GUIPlugin 31
Listing 9-7 Get data from the plug-ins 31
Listing 9-8 Utilities 32

Chapter 10 A Worked Design Example 8-1

Figure 10-1 The Microwave Anisotropy Probe 8-2
Listing 10-1 TMAPSim: File header 8-4
Listing 10-2 TMAPSim: Sampling intervals 8-4
Listing 10-3 TMAPSim: Mass properties 8-5
Listing 10-4 TMAPSim: Tachometer loop design 8-5
Listing 10-5 TMAPSim: ACS design 8-6
Listing 10-6 TMAPSim: Rate loop 8-6
Listing 10-7 TMAPSim: Initialize the control system 8-7
Listing 10-8 TMAPSim: Set up the simulation 8-8
Listing 10-9 TMAPSim: Start of the simulation loop 8-9
Listing 10-10 TMAPSim: Sensor models 8-9
Listing 10-11 TMAPSim: Control system 8-10
Listing 10-12 TMAPSim: Error computation 8-10
Listing 10-13 TMAPSim: Attitude loop 8-11

Listing 10-14 TMAPSim: Compute wheel speed demand 8-11
Listing 10-15 TMAPSim: RWA tach loops 8-11
Listing 10-16 TMAPSim: Simulation 8-12
Listing 10-17 TMAPSim: Saving plot information 8-12
Figure 10-2 TMAPSim: Motion of the unit vector. 8-13
Listing 10-18 TMAPSim: Plotting 8-13
Figure 10-3 TMAPSim: Precession of the unit vector 8-14

Chapter 11 Simulation 9-1

Table 11-1 Attitude Simulation Functions 9-3
Figure 11-1 Rigid body simulation 9-5
Listing 11-1 Rigid Body Attitude Simulation 9-5
Figure 11-2 Wire model 9-6
Listing 11-2 Wire Simulation: Header 9-7
Listing 11-3 Wire Simulation: Initialization 9-7
Listing 11-4 Wire Simulation: Spacecraft properties 9-8
Listing 11-5 Wire Simulation: Initialization of the wire model 9-8
Listing 11-6 Wire Simulation: Kinematic constraints 9-9
Listing 11-7 Wire Simulation: Initialization of plotting 9-9
Listing 11-8 Wire Simulation: Simulation loop plotting 9-10
Listing 11-9 Wire Simulation: Dynamics model 9-10
Figure 11-3 Wire simulation: Results 9-11
Listing 11-10 Wire Simulation: Plotting 9-11
Figure 11-4 Topological Tree 9-12
Listing 11-11 Tree Simulation: Initialization 9-13
Listing 11-12 Tree Simulation: Adding bodies. 9-13
Listing 11-13 Tree Simulation: A hinge acceleration. 9-13
Listing 11-14 Tree Simulation: A low earth orbit. 9-14
Listing 11-15 Tree Simulation: Initialization 9-14
Figure 11-5 Tree Simulation: One of several plots. 9-15
Listing 11-16 Tree Simulation: The simulation loop. 9-15
Listing 11-17 Tree Simulation: Plotting. 9-15
Listing 11-18 NORAD.m 9-16

Chapter 12 Orbit Propagation 19

Figure 12-1 Orbit Propagation GUI 21
Table 12-1 Element sets 23
Table 12-2 Stopping Conditions 26
Table 12-3 PropagateOrbitPlugin API 29
Table 12-4 Plugin Functions 30
Figure 12-2 `FEngine` GUI 31

Figure 12-3 `FAero` GUI 31
Figure 12-4 `FSolar` GUI 32
Figure 12-5 `FPlantNoise` GUI 33
Listing 12-1 `FAtmDensity` GUI 34
Table 12-5 Jacchia J70 Atm inputs. 34
Listing 12-2 FAero: Main Function 35
Listing 12-3 `FAero`: `Initialize` 36
Listing 12-4 `FAero: CreateCallback` 37
Listing 12-5 `FAero`: Getting and setting data. 37
Listing 12-6 `FAero`: The `Update` subfunction. 37
Table 12-6 I/O for STKOrbit 39

Chapter 13 Budgets 10-1

Listing 13-1 PBudget Example 10-2
Listing 13-2 Fuel Budget: Moment Arms 10-3
Listing 13-3 Fuel Budget: Thruster performance data 10-4
Listing 13-4 Fuel Budget: General spacecraft data 10-4
Listing 13-5 Fuel Budget: Array initialization 10-5
Listing 13-6 Fuel Budget: Continuous delta-v burn 10-5
Listing 13-7 Fuel Budget: Delta-v with off-pulsing 10-6
Listing 13-8 Fuel Budget: Unloading momentum 10-6
Listing 13-9 Fuel Budget: Using ACS thrusters 10-7
Listing 13-10 Fuel Budget: Firing a few pulses 10-7
Listing 13-11 Fuel Budget Example 10-7

CHAPTER 1

Installing The Spacecraft Control Toolbox

This chapter shows you how to install the The Spacecraft Control Toolbox (SCT). It also shows you how to control the path to the Spacecraft Control Toolbox functions.

Requirements

MATLAB 5.2 is required to run all of the functions. Most of the new functions use data structures and cell arrays and are not compatible with versions of MATLAB prior to 5.0.

Your monitor should have a resolution of at least 800 by 600 pixels. Color is desirable.

The Spacecraft Control Toolbox does not use any functions that were added after V5.2 of MATLAB.

About this Tutorial

This tutorial uses the following conventions

www.psatellite.com	URL's and email addresses. The first letter is always lower case. Beginnings of words after the first are always upper case.
`monoSpace`	MATLAB code.
button	GUI button

The examples in this tutorial were run on a Macintosh PowerPC computer. All pictures of GUIs were created using Macintosh screen shots and reduced 50% for inclusion in this tutorial.

All web pages were captured using Netscape 4.5.

Background

The Spacecraft Control Toolbox is a set of MATLAB.m files. All of the functions are in m-file format. No mex files or.p (parsed) files are supplied, As a consequence you have complete access to all of the source code.

You can obtain the software in two ways. One is on CD-ROM. The second way is by downloading the software from the Princeton Satellite Systems website at www.psatellite.com.

CD-ROM

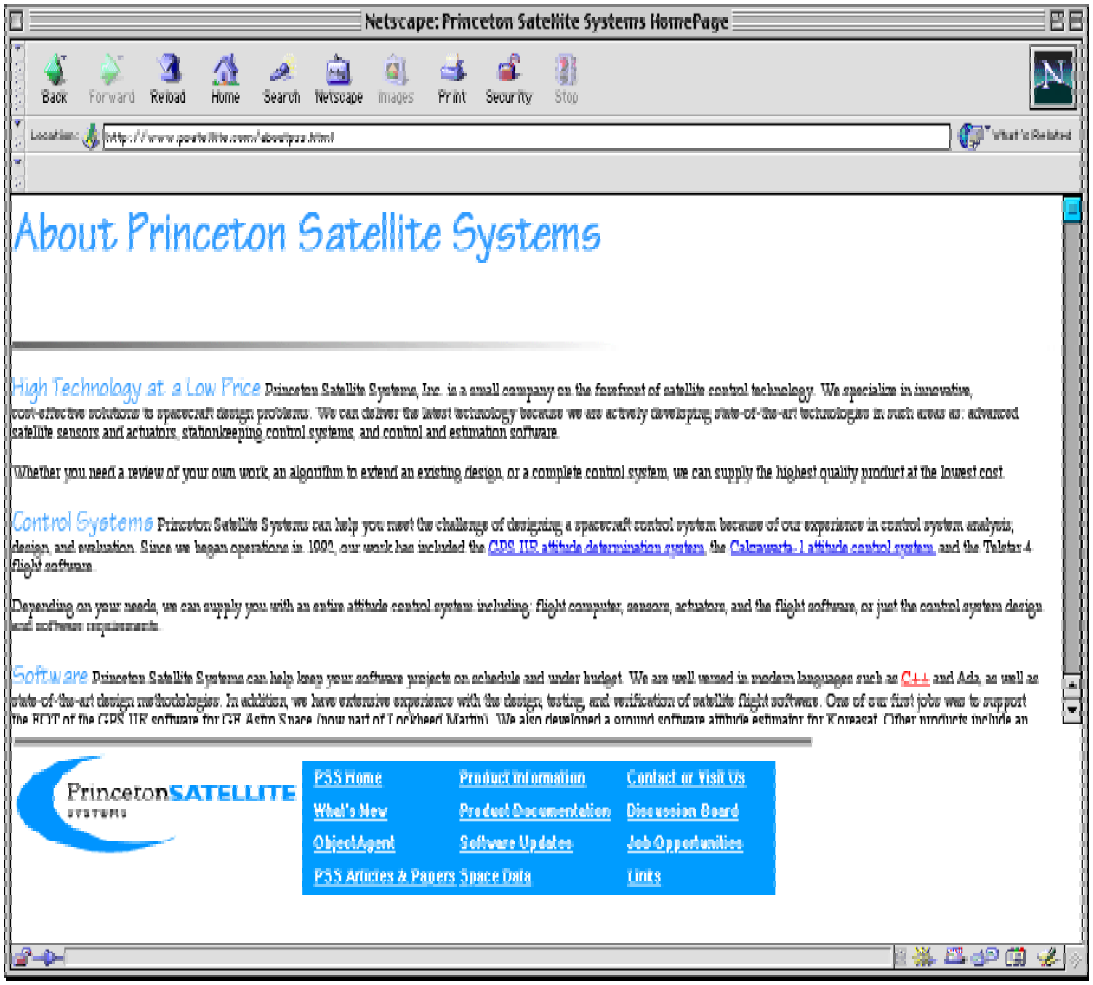
The software is supplied on a CD-ROM with two partitions. One is a Macintosh Hierarchical File System (HFS) partition and the other is an ISO standard partition. The latter uses Macintosh file naming conventions so is not truly ISO compatible. Macintosh files are mixed case and can have up to 32 characters. It is important that the mixed case not be changed (which sometimes happens with installers) because the software is case-sensitive.

Some systems, such as Silicon Graphics UNIX machines, by default will see the HFS partition. Most will see the ISO partition.

Web

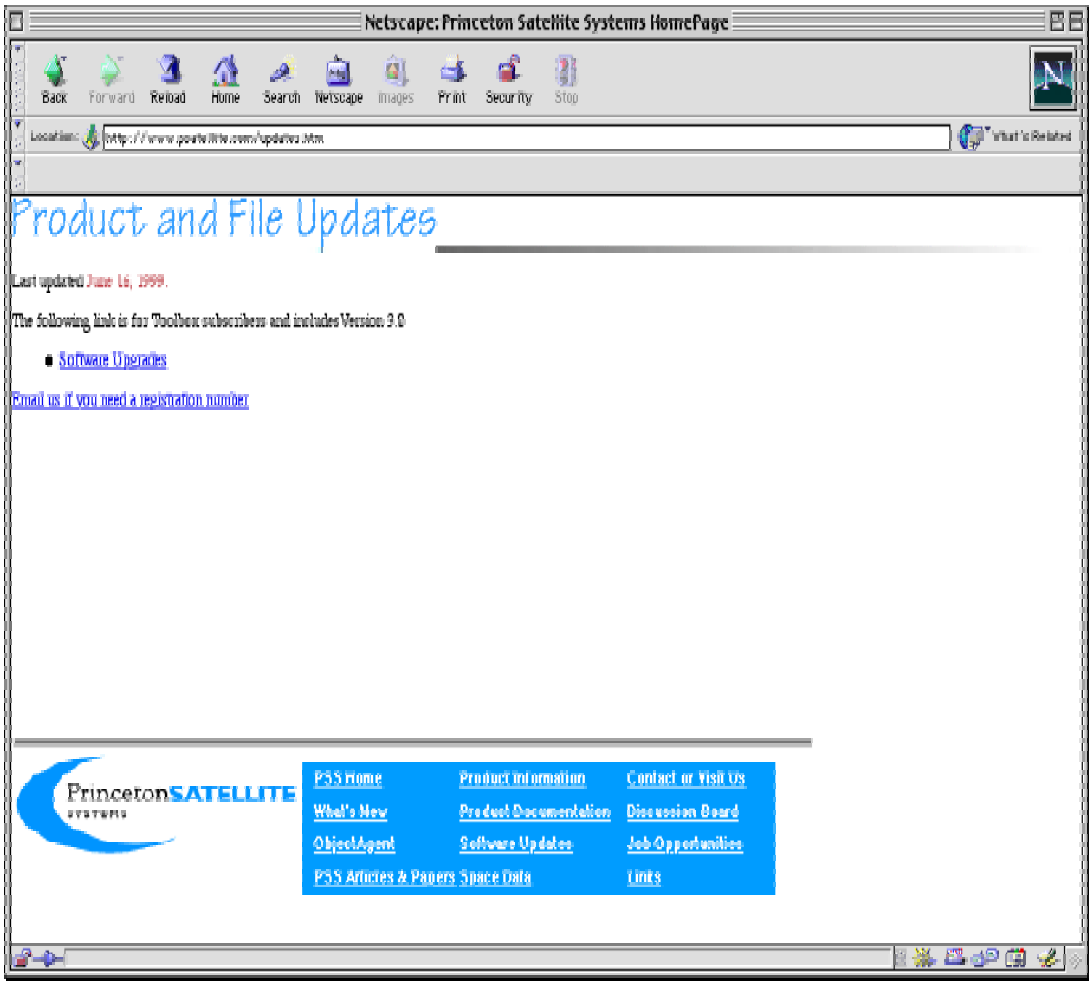
First, get to the PSS website at http://www.psatellite.com. The website uses frames. The frame at the bottom of the webpage allows you to navigate throughout the site. Click on the upgrades button.

Figure 1-1 Home page for the PSS website



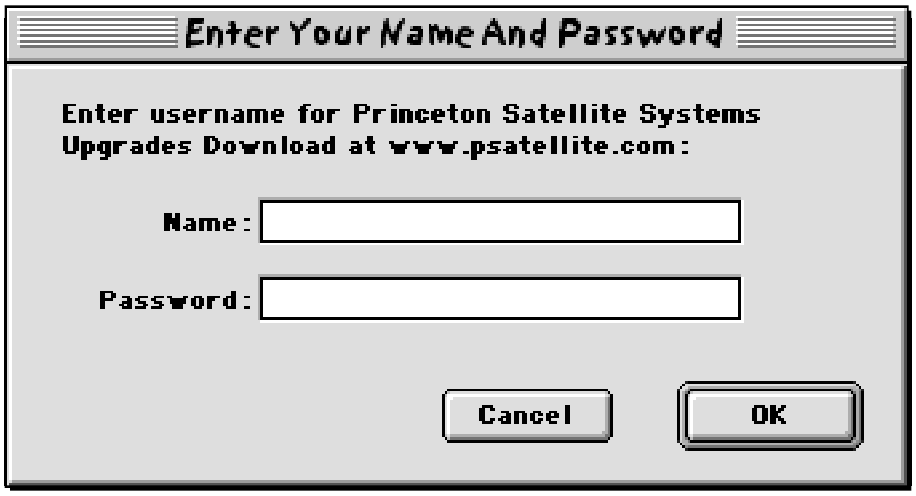
This will take you to the upgrades page shown in Figure 1-2 on page 5.

Figure 1-2 The upgrades page



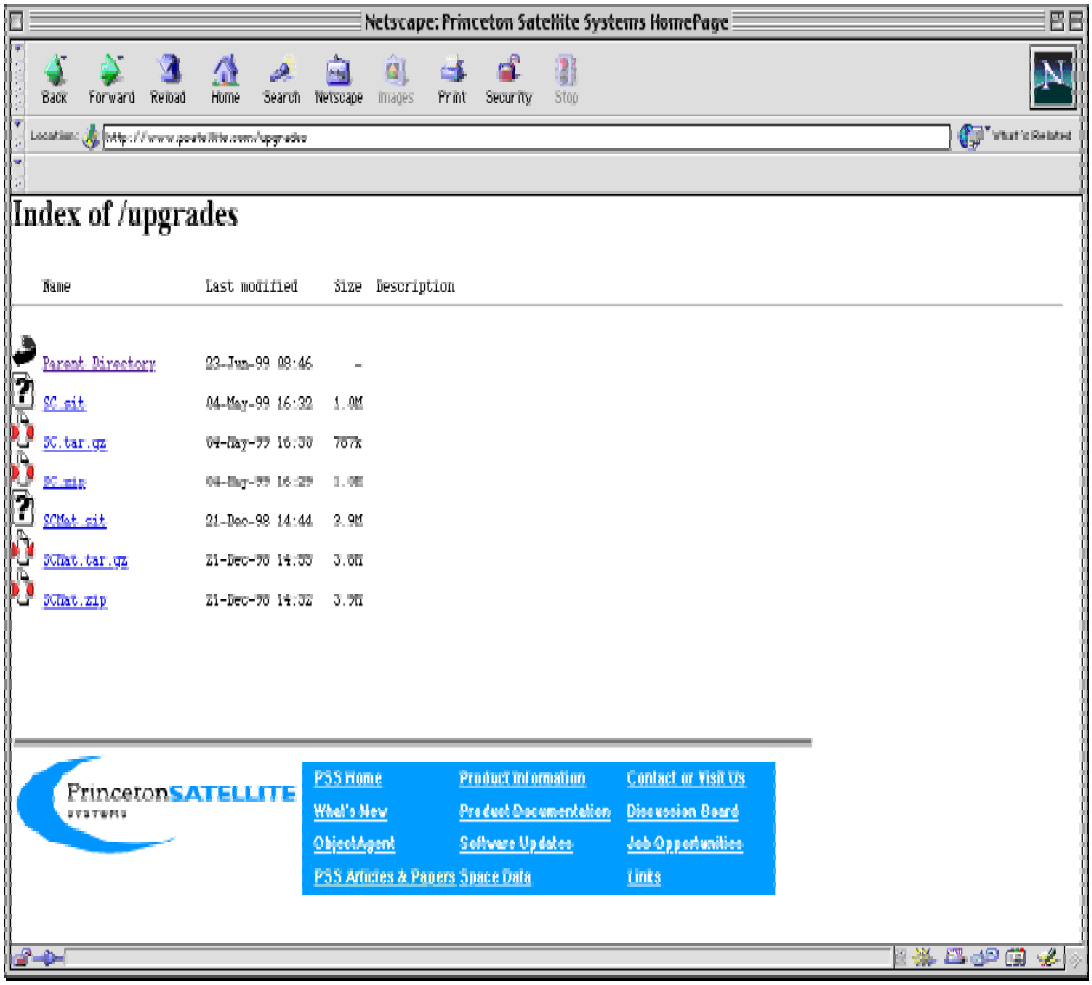
Click on Software Upgrades. A dialog box will appear.

Figure 1-3 Password and user id



Enter your user id and password and click ok. The same dialog will appear. Enter the data again. You will then see the upgrades list.

Figure 1-4 Upgrades



All of the files are stored in archives. The files and their types are listed in

Table 1-1 Archives

File	Macintosh	Windows	UNIX
Toolbox Functions	SC.sit	SC.zip	SC.tar.gz
.mat Files	SCMat.sit	SCMat.zip	SCMat.tar.gz

Table 1-1 on page 7. Download all of the files in the appropriate column. Your net browser should decompress the files. You need version 5.1 of the Stuffit decompression utility on the Macintosh to decompress the.sit files.

Included Files

The files listed in Table 1-2 on page 8.

Table 1-2 Files included in the Spacecraft Control Toolbox

Files	Explanation
SC	Directory of toolbox directories
catalog.pdf	Catalog of PSS products
tutorial.pdf	This tutorial
license.pdf	The software license. Please read it!

The documentation is readable using Acrobat Reader 3.x or greater. If you downloaded the files, copy SCMat into the SC directory.

Installation

Copy the directory into any convenient location on your system. Start Matlab and find the function PSSInstaller. Type in the command window:

```
PSSInstaller
```

This will set the paths.

Case Sensitivity

SCT files are case sensitive. All SCT filenames use the convention that the first letter is capitalized and word breaks (even if the word is represented by an initial) are capitalized. For example you should see:

```
RVOrbHF.m
```

not

```
rvorbhf.m
```

If you see the latter either contact your system administrator or send email to support@psatellite.com. If you see the latter and copied the files from a CD-ROM you may have a problem with you copy utility or the CD has a problem. If you downloaded from the web, your decompression utility preferences may have to be changed.

Function Overloading

The Spacecraft Control Toolbox overloads the MATLAB `Cross` and `Dot` functions and the `earth.mat` file. The SCT functions have all the functionality of the MATLAB `Cross` and `Dot` functions but have additional features. The SCT `earth.mat` file is a special data structure used by the toolbox for displaying planetary images. If you have the SCT in front of the other MATLAB toolboxes this will not present a problem. However, some SCT functions will not work properly with the MATLAB `Cross` and `Dot` functions.

Updates

You will receive email notification of updates. The email will tell you which files need to be downloaded. If you cannot download the files contact support@psatellite.com and we will send you a new CD.

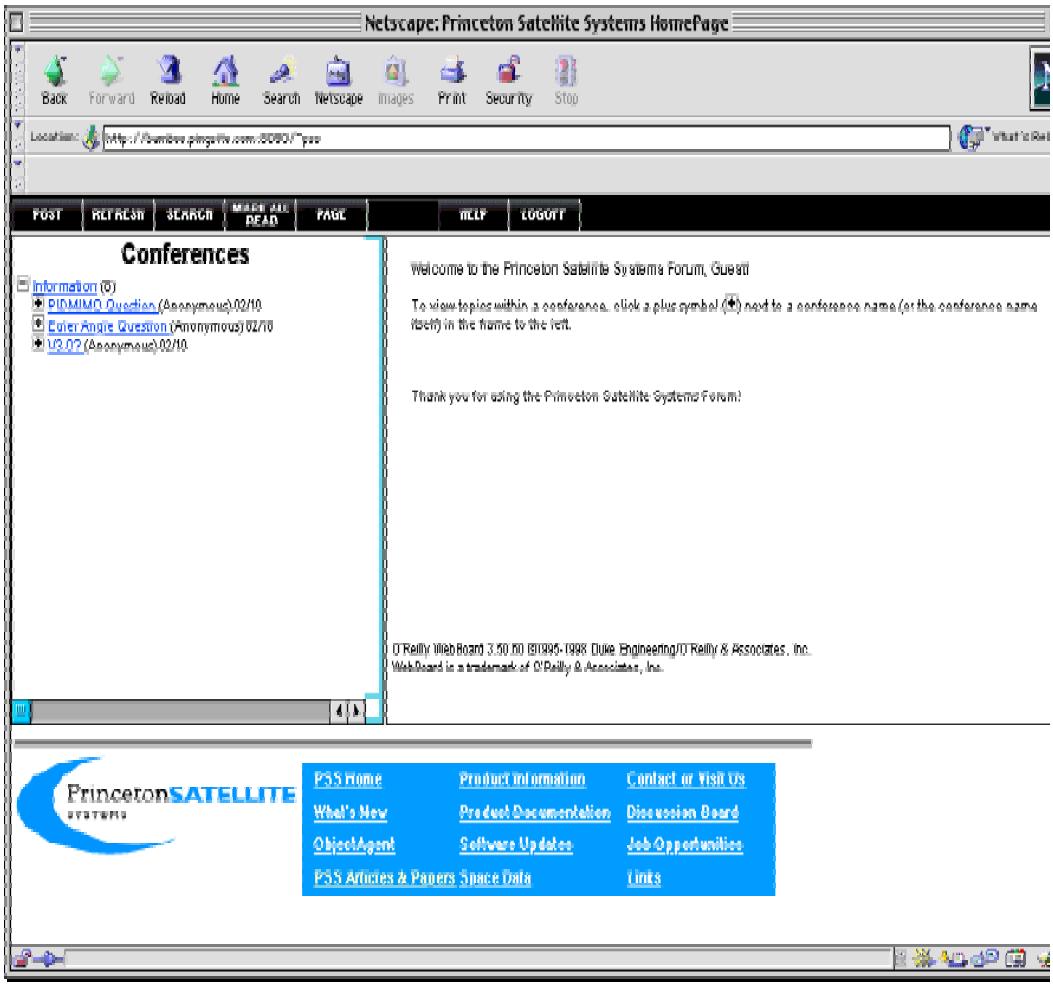
Technical Support

Contact support@psatellite.com for technical support.

Webboard

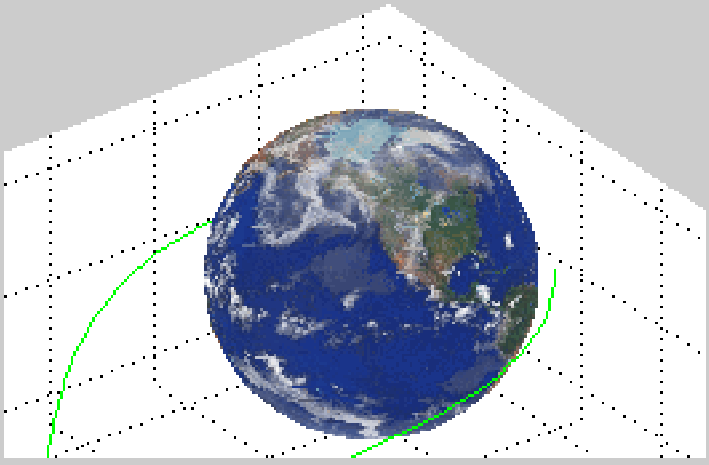
You can talk with fellow SCT toolbox users and PSS engineers using our Webboard. Go to http://www.psatellite.com and hit Discussion Board. You will then see the webboard.

Figure 1-5 Webboard



CHAPTER 2

Getting Help

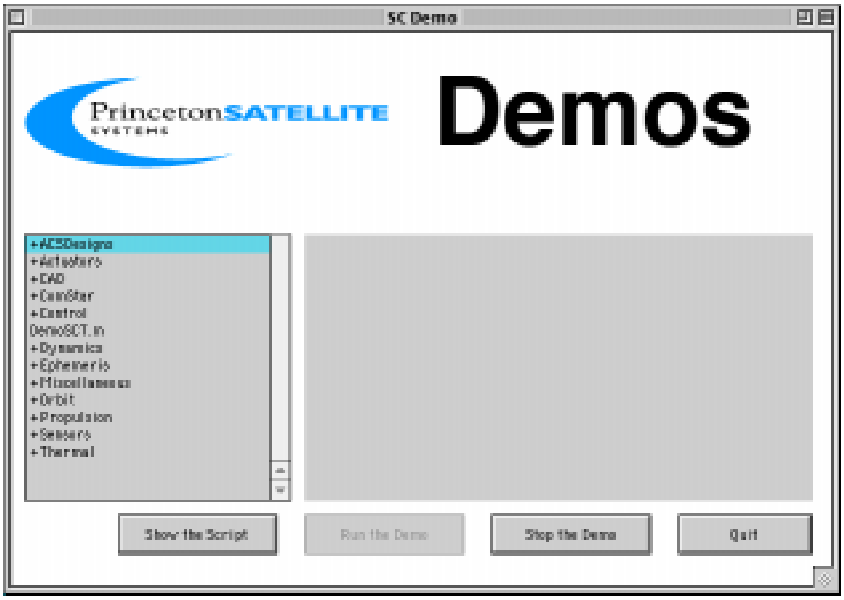


This chapter shows you how to use the help systems built into the Spacecraft Control Toolbox. There are three sources of help. One is the demo functions, the second is the file help function and the third is the graphical user interface help system.

Demos

If you type `DemoSC` you will see the following GUI.

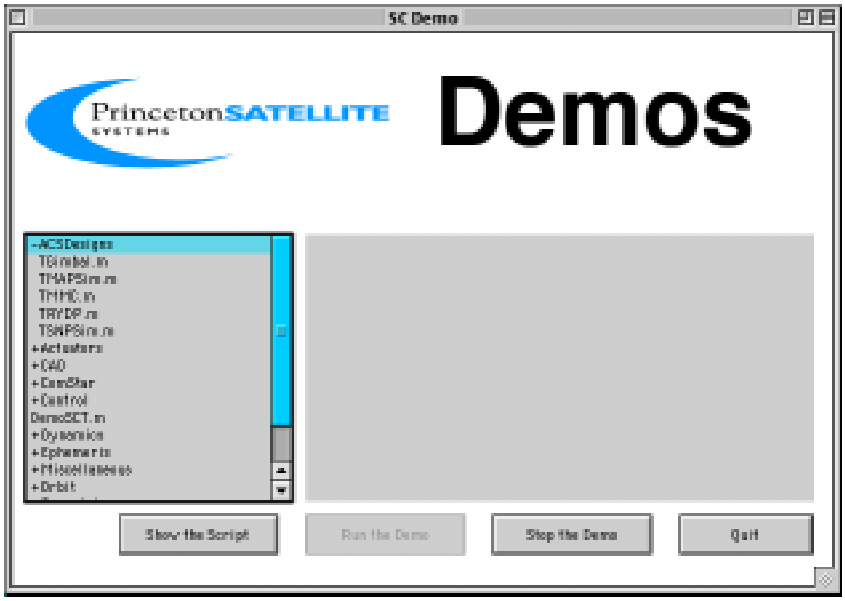
Figure 2-1 DemoSC



The list on the left-hand-side is hierarchical. The GUI checks to see which directories are in the same directory as `DemoSC` and lists all directories and files. This allows you to add your own directories and demo files.

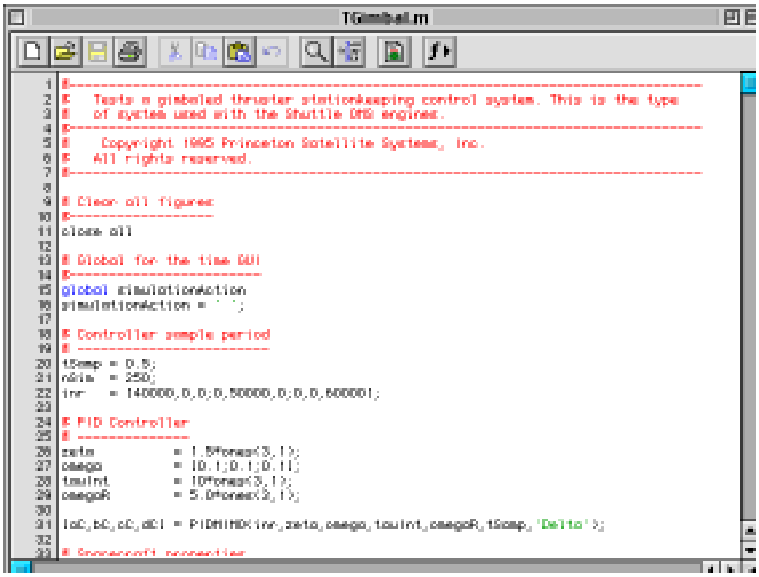
The first category is `ACSDesigns`. Click on the name to open the directory. The + sign changes to − and the list changes.

Figure 2-2 Opening ACSDesigns



If you would like to look at, or edit, the script, hit "`Show the Script`." This opens the MATLAB edit window.

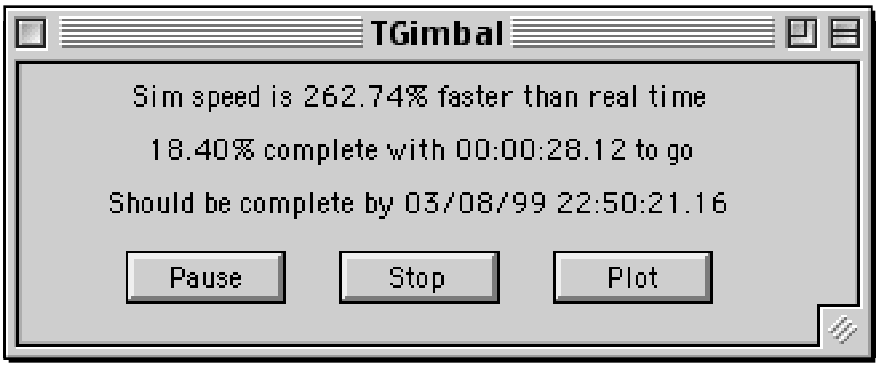
Figure 2-3 Edit Window


TGimbal.m
% Tests a gimbaled thruster stationkeeping control system. This is the type
% of system used with the Shuttle OMS engines.
% Copyright 1995 Princeton Satellite Systems, Inc.
% All rights reserved.
% Clear all figures
close all
% Global for the time GUI
global simulationAction
simulationAction = ' ';
% Controller sample period
tSamp = 0.5;
nMax = 250;
inr = 140000,0,0;0,50000,0;0,0,60000];
% PID Controller
zeta = 1.5*ones(3,1);
omega = [0.1;0.1;0.1];
tauInt = 10*ones(3,1);
omegaR = 5.0*ones(3,1);
[cC,sC,cC,dCl] = PIDMIMO(inr,zeta,omega,tauInt,omegaR,tSamp,'Delta');

Select `TGimbal` and hit "`Run The Demo.`" The following time display will appear. It shows the ratio of simulation speed to real-time, gives the per-

centage completed and gives you an estimate of when the run will be done. You can pause

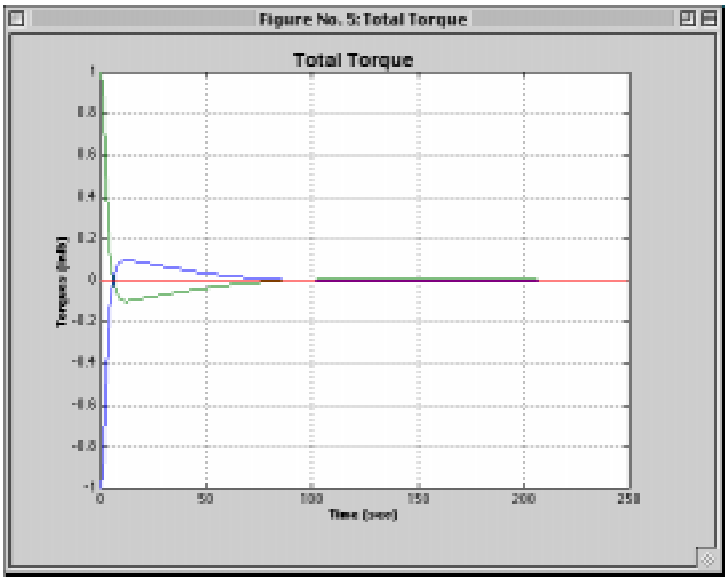
Figure 2-4 Time-to-Go Display



the display, stop the simulation or plot the current results using the buttons in the GUI.

If you let it run to completion, several plots will appear. The following is the last plot.

Figure 2-5 Results of `TGimbal`



`DemoSC` has a wide range of demonstrations. Not all use the Time GUI shown above. These cannot be stopped using the GUI controls but all of these are short.

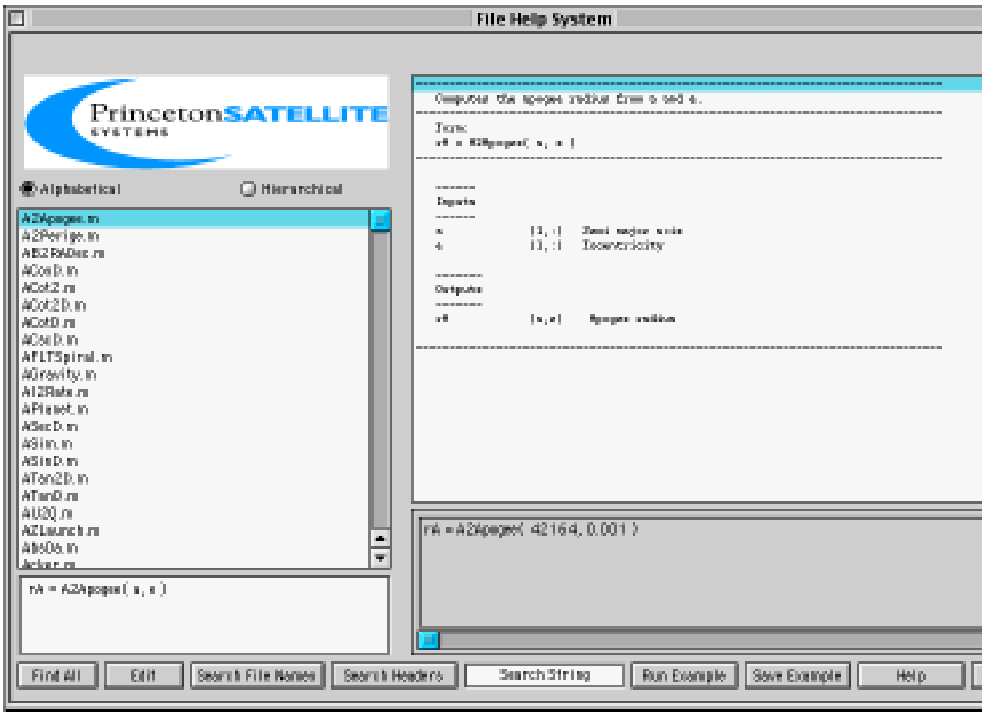
File Help

Introduction

The FileHelp function gives you access to the headers of all of the functions in the Spacecraft Control Toolbox. You can browse the headers and try out examples associated with each function. You can also edit the examples, create new examples and save them into the help database.

When you type FileHelp the FileHelp GUI appears.

Figure 2-6 The file help GUI

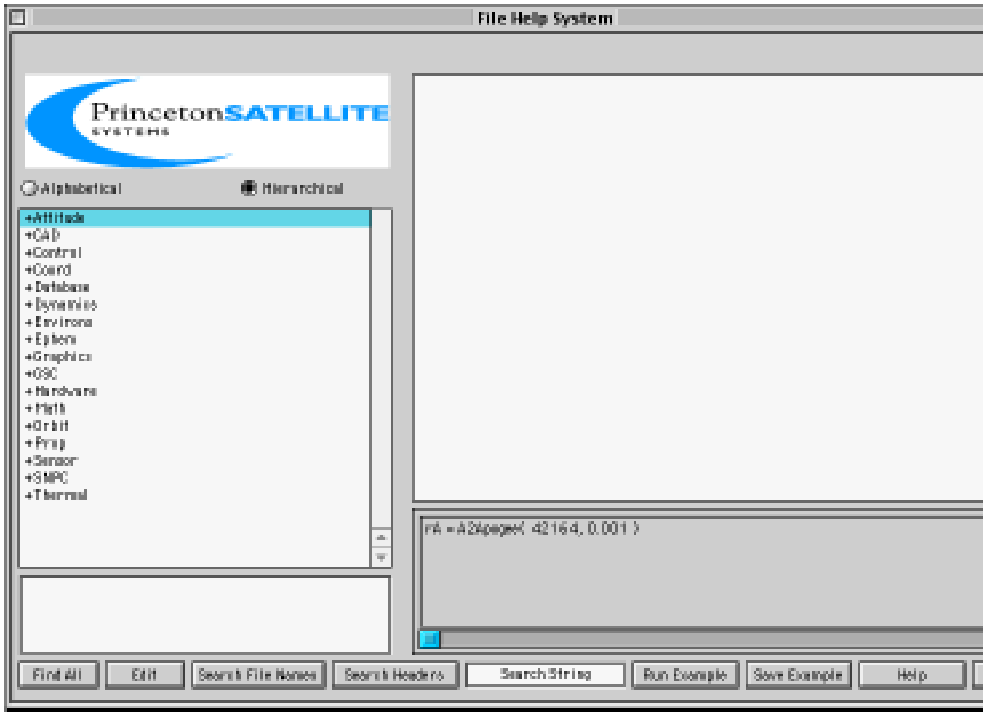


There are four main panes in the window. On the left hand side is a display of all files in the toolbox. The list initially is alphabetical. On the right-hand-side is the header of the first file. Immediately below the header display is the editable example. To its left is a template for the function. You can cut and past it into your script our function.

The List Pane

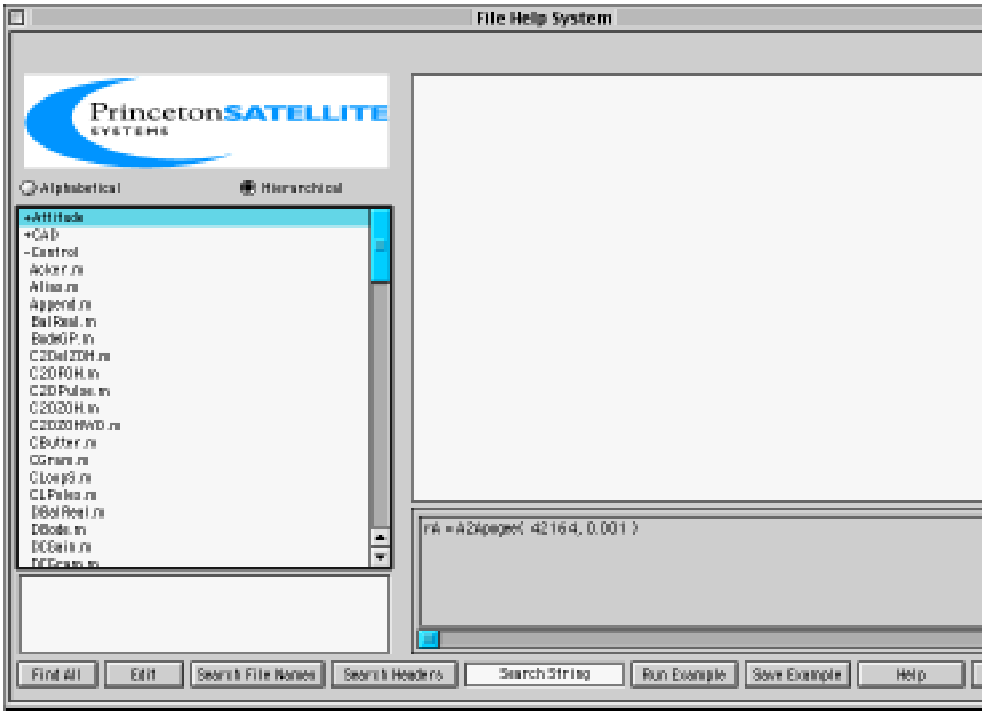
Above the list pane are two buttons. If you hit hierarchical, the names of the SCT directories appears.

Figure 2-7 Hierarchical file display



A + sign is in front of each name. Click the file with a plus sign to see which toolbox functions are in each directory. To view a file's header, example and template, click on the file name.

Figure 2-8 Open directory



This list is of the toolbox as you receive it on CD-ROM or as downloaded from the web. It will not show any of your functions. A build script is included if you want to customize this help display.

If you toggle from hierarchical to alphabetical and back you will notice that the open directories stay open in the hierarchical display.

The Example Pane

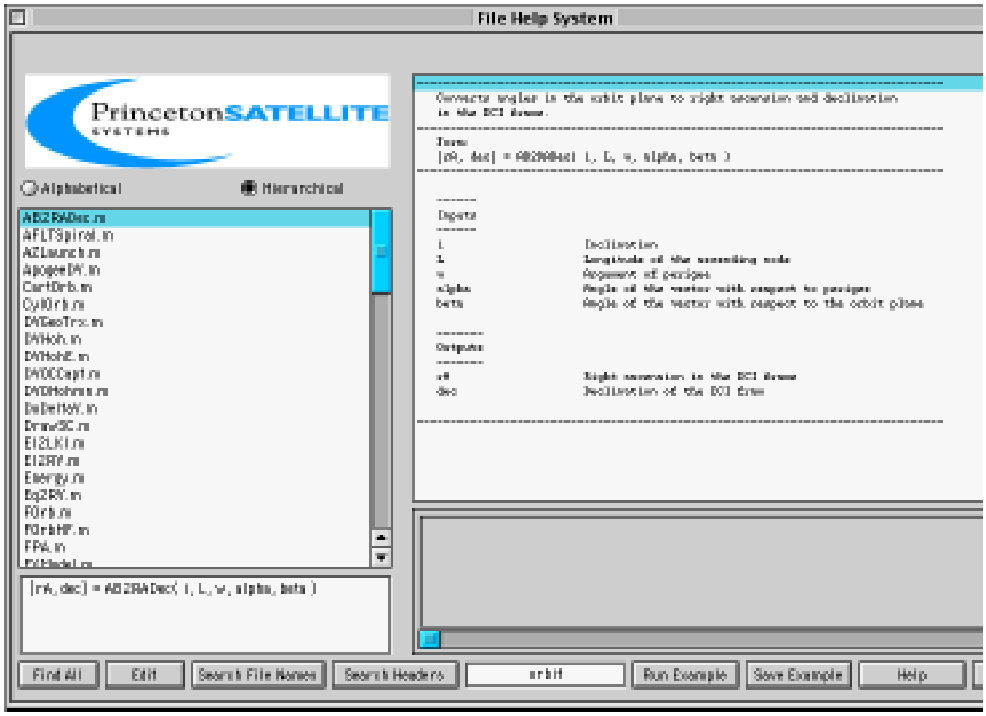
This pane gives an example for the function displayed. Not all functions have examples. The edit display has scroll bars. You can edit the example, create new examples and save them using the buttons below the display. To run an example hit the “Run Example” button.

You can include comments in the example by using the% symbol.

Find All Button

Find all returns to the list of all of the functions. This is used after one of the search options has been used. `Find All` will return to either an alphabetical or hierarchical list, depending on which is selected above the file pane.

Figure 2-9 Find all



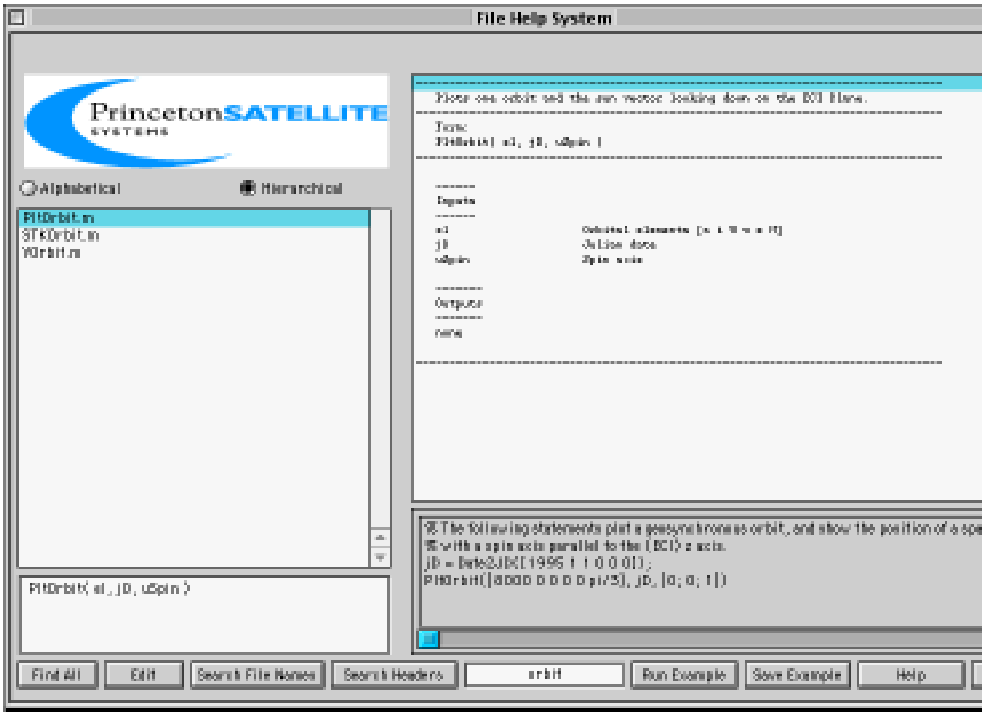
Edit Button

This opens the MATLAB edit window for the function selected in the list.

Search File Names Button

Search file names for a string. This function looks for exact, but not case sensitive, matches. The file display displays all matches.

Figure 2-10 Find “orbit” in the file names

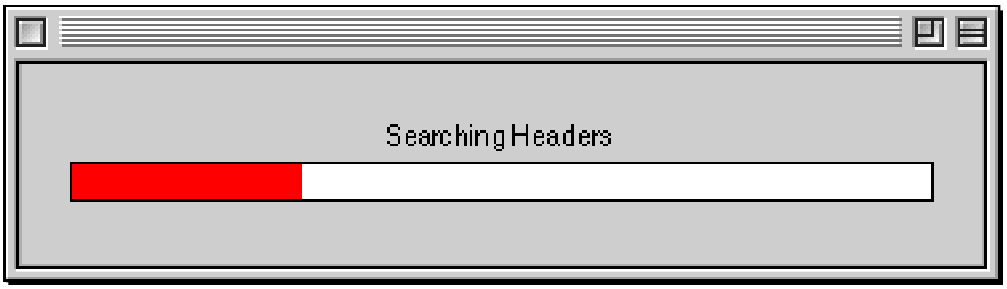


Search Headers Button

Search headers for a string. This function looks for exact, but not case sensitive, matches. The file display displays all matches.

A progress bar gives you an indication of time remaining.

Figure 2-11 Progress bar



Search String Edit Box

This is the search string. Spaces will be matched so if you type “attitude control” it will not match “attitude control” (with two spaces.)

Run Example Button

Run the example in the display. Some of the examples are just the name of the function. These are functions with built-in demos.

Results will appear either in separate figure windows or in the MATLAB Command Window.

Save Example Button

Save the example in the edit window. Hitting this button only saves it in the temporary memory used by the GUI. You can save the example permanently when you Quit.

Help Button

Open this help system.

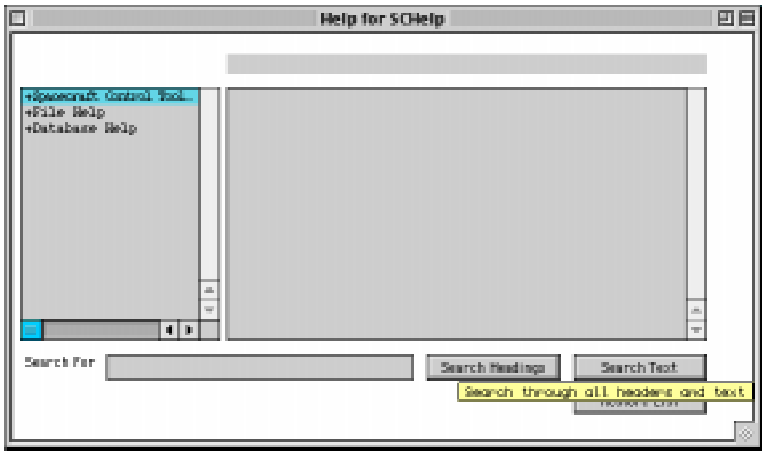
Quit

Quit the GUI. If you have edited an example, it will ask you whether you want to save the example before you quit.

Graphical User Interface Help

Each graphical user interface (GUI) has a help button. If you hit the help button a new GUI will appear.

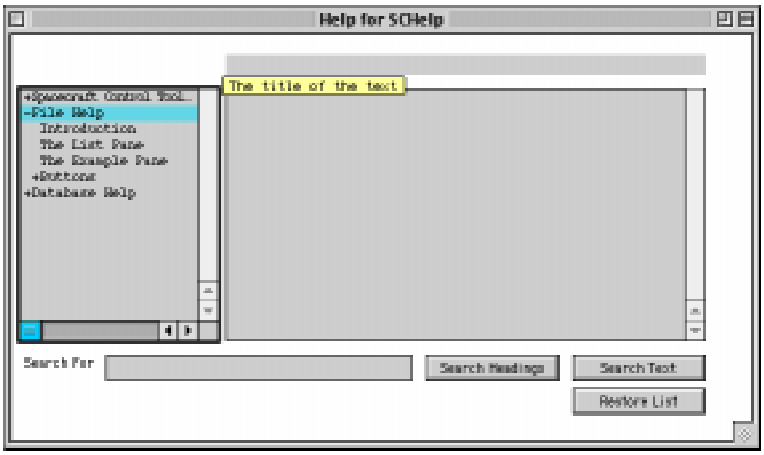
Figure 2-12 SCHelp GUI



You can access on-line help about any of the GUIs through this display. It is separate from the file help GUI described above.

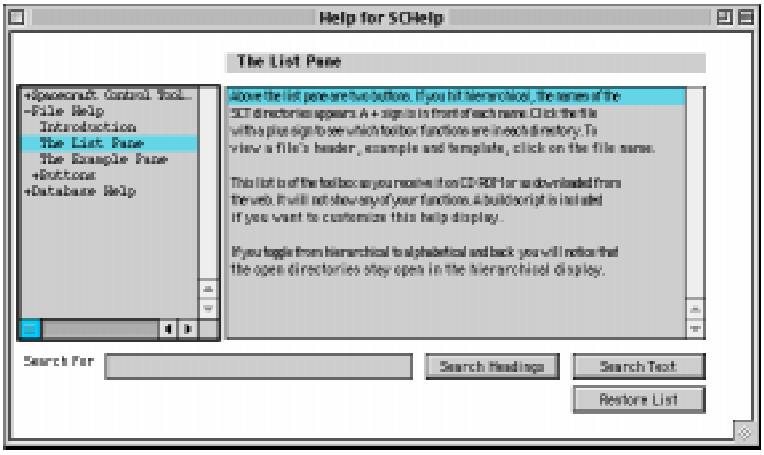
The display is hierarchical. Any list item with a + or − in front is a directory. + means the directory list is closed, − means it is open. If you click on one of the list elements you get the display shown in Figure 2-13 on page 11.

Figure 2-13 An open directory in SCHelp



If you click on any other name in the list you will get a text display in the right-hand pane.

Figure 2-14 Help



You can either search the headings or the text by entering a text string into the "`Search For`" edit box and hitting the appropriate button. "`Restore List`" restores the list window to its previous configuration.

MATLAB Help

You can see the header for any file by typing

```
help fileName
```

in the MATLAB command window.

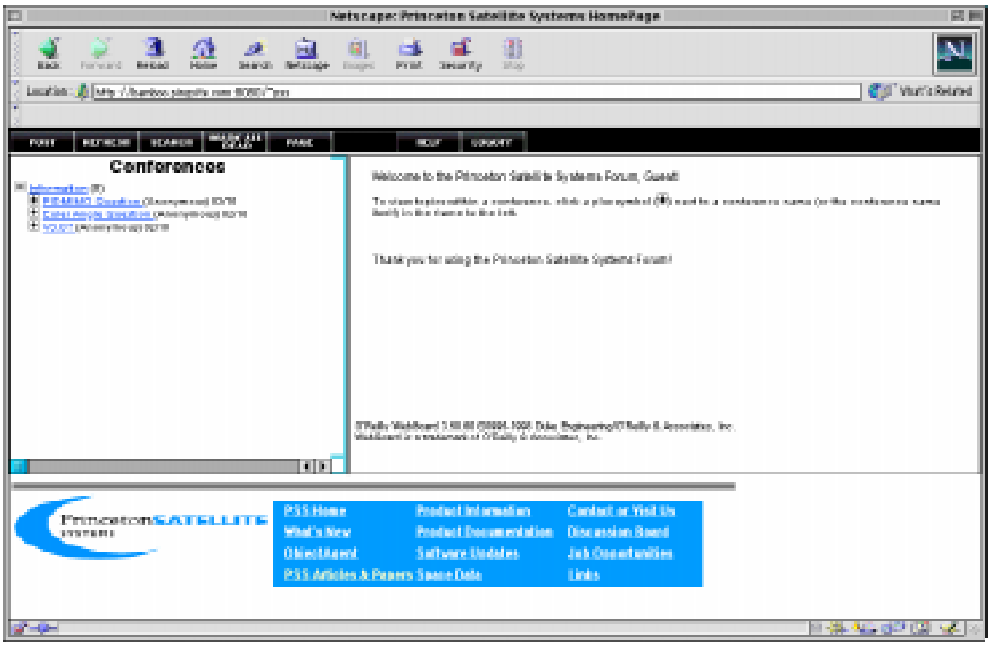
Technical Support

Contact support@psatellite.com for technical support.

Webboard

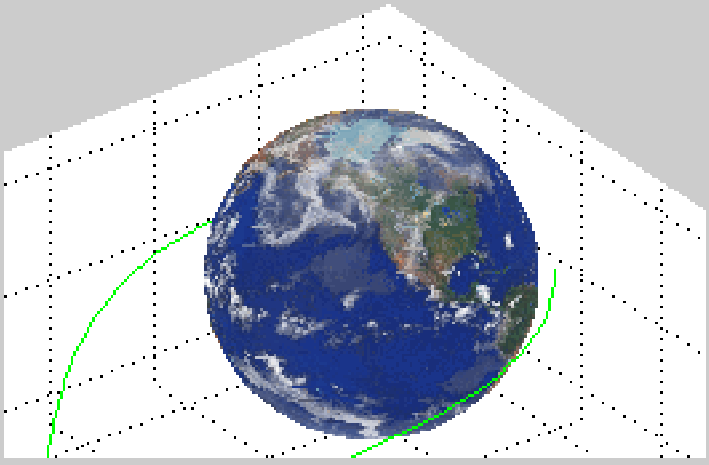
You can talk with fellow SCT toolbox users and PSS engineers using our Webboard. Go to http://www.psatellite.com and hit Discussion Board. You will then see the webboard.

Figure 2-15 Webboard



CHAPTER 3

Using Databases



This chapter shows you how to use the database and constant functions in the toolboxes. These functions allow you to manage the various constants and parameters used in your projects and insure that all of your engineers are using consistent numbers in all their analyses.

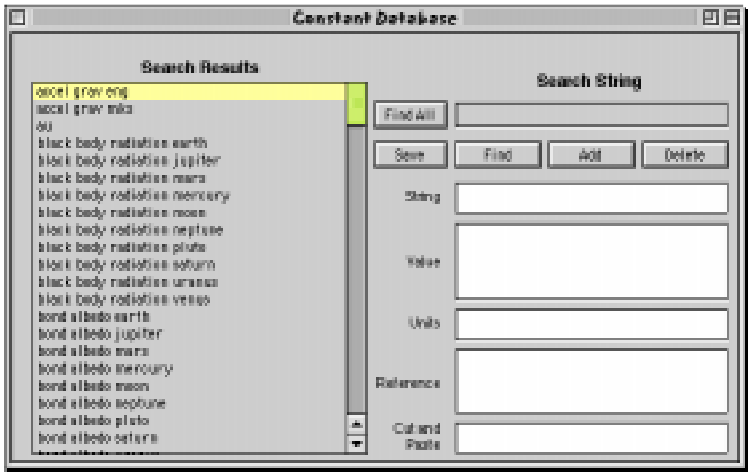
The Constant Database

The constant database gives a substantial selection of constants that are useful for the aerospace engineer. If you type

```
Constant
```

you will get the following GUI

Figure 3-1 Constant database

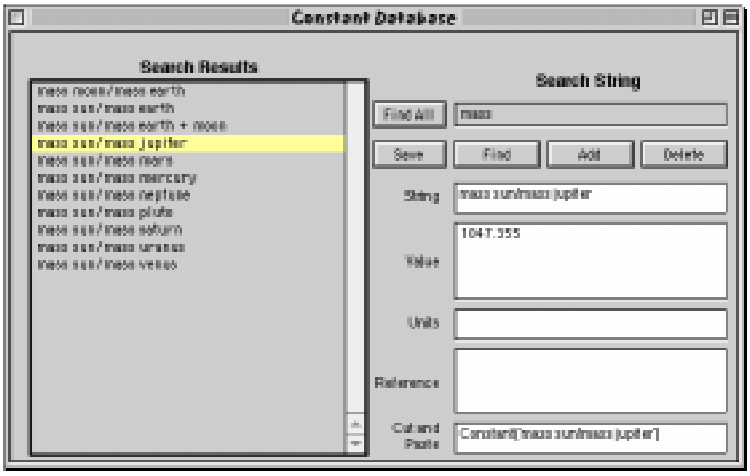


The list on the left is a list of all of the constants in the database. You can enter a search string and look for matches by hitting Find. If you then click one of the selections the GUI looks like it does in the following figure.

This function always loads its constants from the `.mat` file

```
SCTConstants.mat
```

Figure 3-2 Searching for `'mass'`



The string field shows the parameter name. Directly below it is the value for the parameter. The value may be any MATLAB construct. Directly below that is a field for units, then a field for reference information and finally a field that gives a template for the function. You can cut and paste this into any function or script.

Searches are case and whitespace insensitive.

You can add new constants. Just type a constant name in the `String` field, a value in the `value` field and optionally, units and reference information. Hit Add. You will get a warning if you try and replace an existing constant.

You always hit Add even if you are modifying the value of an existing constant.

You can delete a constant by hitting the Delete button.

You can access the database through the command line. Just type:

```
Constant('mass sun/mass jupiter')

ans =

   1.0474e+03
```

The database loads its constants from a database the first time it is launched. Once it is launched it will not load a new database, so there is relatively little overhead using it in your scripts and functions.

Merging Constant Databases

Periodically, PSS, will release new constant databases. If you have customized your own database you can merge it with the PSS database using the `MergeConstantDB` function. Just type

```
MergeConstantDB( 'initialize', a, b )
```

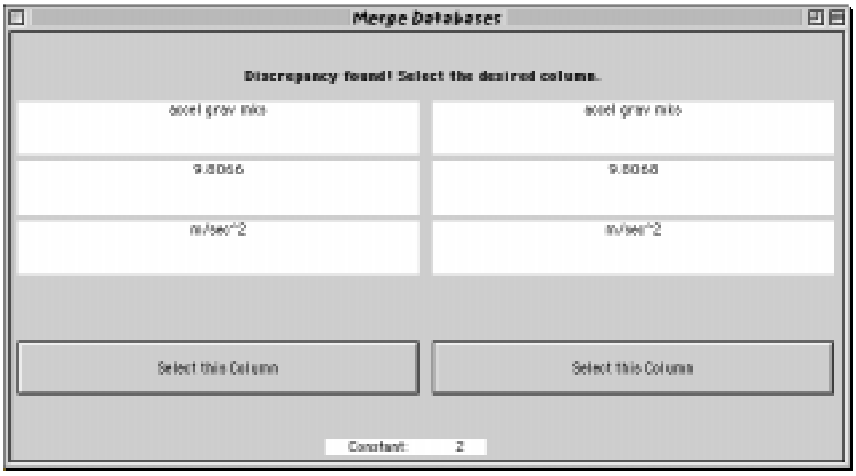
where `a` and `b` are the.mat files to be merged. The standard PSS database is called `SCTConstants.mat`.

In this example we have modified the value of `'accel grav mks'` to be 9.8068. We type

```
MergeConstantDB('initialize','SCTConstants.mat','SCT
ConstantsOld.mat')
```

You will see the following display.

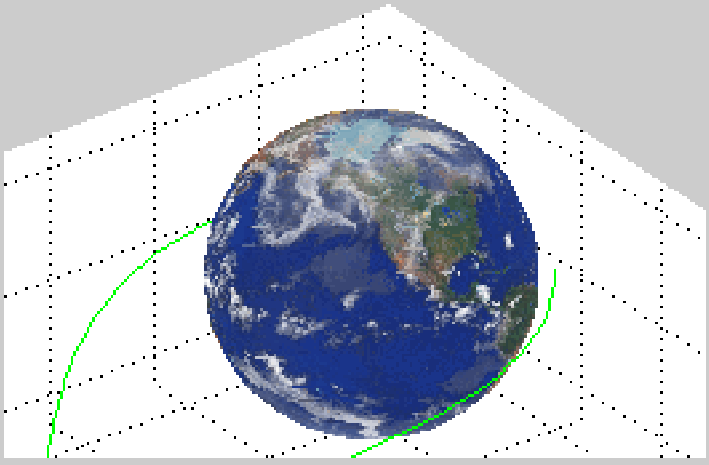
Figure 3-3 MergeConstantDB function



Just click the button for the column you wish to include in `'SCTConstants.mat'`.

CHAPTER 4

Basic Functions



This chapter shows you how to use a sampling of the Spacecraft Control Toolbox functions.

Introduction

The Spacecraft Control Toolbox is composed of over five hundred Matlab m-files. The functions cover attitude control and dynamics, orbit dynamics and kinematics, ephemeris, thermal and mathematics operations. Most of the functions can be used individually although some are rarely called except by other toolbox functions.

The SPACECRAFT CONTROL TOOLBOX is organized into a number of directories based on topic. These directories organize the on-line help feature in MATLAB.

Currently, the toolbox is broken into the directories listed in the following table. Several of the directories are new in V3.0.

Table 4-1 Toolbox Organization

Directory	Purpose
Actuator	Actuator models.
Attitude	Attitude control analysis functions.
CAD	Compute aided design.
Classes†	MATLAB classes for state space systems, quaternions and rotation matrices.
Control	Control system design functions.
Coord	Coordinate transformation.
Database	Database examples.
Dynamics	Functions for numerically simulating spacecraft.
Environs	Environmental models including magnetic field, atmospheric density, etc.
Ephem	Ephemeris functions giving orbital elements of the planets, earth orientation, etc.
† New in V3.0	

Table 4-1 Toolbox Organization

Directory	Purpose
General†	General purpose functions.
Graphics	Drawing plots, 3D pictures, etc.
GSC	Functions implementing a geosynchronous satellite control system.
GUIPlugin†	Graphical user element plug in elements.
Hardware	Miscellaneous hardware models.
Help	Help system functions.
LowLevel	Functions used by other functions. Will all be eliminated in a future version of the toolbox.
Math	Math functions.
Orbit	Orbit analysis and dynamics functions.
Prop	Propulsion system models.
SCData	Data used by the functions.
Scripts	Example scripts.
Sensor	Sensor models.
SimulinkIF†	Interface function for Simulink.
SNPC	Functions used to implement a sun nadir pointing control system.
Thermal	Thermal analysis functions.
† New in V3.0	

Function Features

Functions have several features that are helpful to understand when using them. Features that are available in the functions are listed in

Table 4-2 Features in SCT functions

Features
Built-in demos
Default parameters
Built-in plotting
Error checking
Variable inputs

These are illustrated in the examples given below.

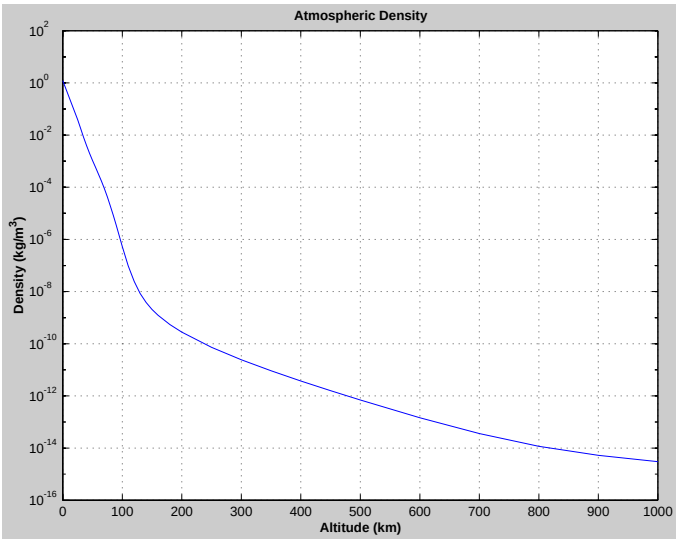
Built-in demos

Many functions have built in demos. A function with a built-in demo requires no inputs and produces a plot for a range of input parameters to give you a feel for the function. An example of a function with a built-in demo is

```
AtmDens2
```

which generates the following plot

Figure 4-1 Atmospheric density from `AtmDens2`



Default parameters

Most functions have default parameters. There are two ways to get default parameters. If you pass

```
[]
```

as a parameter the function will use a default parameter. This is only necessary if you wish to use a default for one parameter and input the value for the next input. For example

```
g = EarthRot( [], 1 )

g =
   -0.8091    0.5877         0
   -0.5877   -0.8091         0
         0         0    1.0000
```

Built-in plotting

Many of the functions in the toolbox will plot if there are no output arguments. In many cases, you do not need any input arguments to get useful plots due to built in defaults.

Most of the functions in the toolbox are compatible with MATLAB 4.x or earlier. However, many of the newer functions make extensive use of data structures and are compatible with versions 5.x or newer. We recommend that you

get the latest version of MATLAB since in the future we will make even more extensive use of data structures and other object oriented features.

The following function will generate a plot.

```
PlanckL
```

and so will

```
32 - PlanckL
```

because there is no equal sign. If no inputs are given it automatically computes a range of values. To prevent plotting (and to get an output) you must type

```
PlanckL(5760,1)
```

or

```
x = PlanckL
```

Error checking

Many functions perform error checking. However, functions that are designed to be called repeatedly, for example the right-hand-side of a set of differential equations tend not have error checking since the impact on performance would be significant. In that case, if you pass it invalid inputs you will get a MATLAB error message.

Variable inputs

Some functions can take different kinds of inputs. An example is Date2JD. You can pass it either an array

```
[  year month day hour minute seconds ]
```

or the data structure

```
d.month
d.day
d.year
d.hour
d.minute
d.second
```

Example Functions

Attitude

Stability for a dual spin spacecraft in geosynchronous orbit. Many commercial satellite are dual-spin stabilized since such a design is very robust.

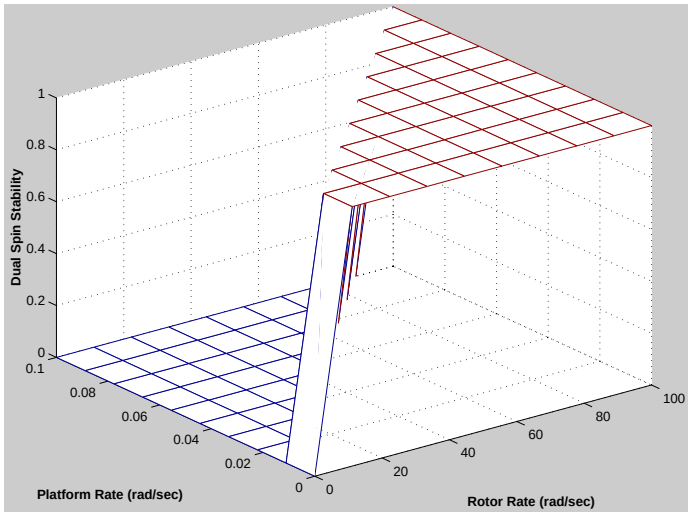
```
dSS = DSpnStab( 1, 100, 1000, 1000, 0, 7.291e-5 )
DSpnStab( 1, 100, 1000, 1000 )
```

The result is

```
dSS =

     0
```

which means the system is unstable! A stability plot is shown below. One means stable. DSpnStab is a typical function in SCT. If no outputs are specified it generates a plot

Figure 4-2 Dual Spin Stability

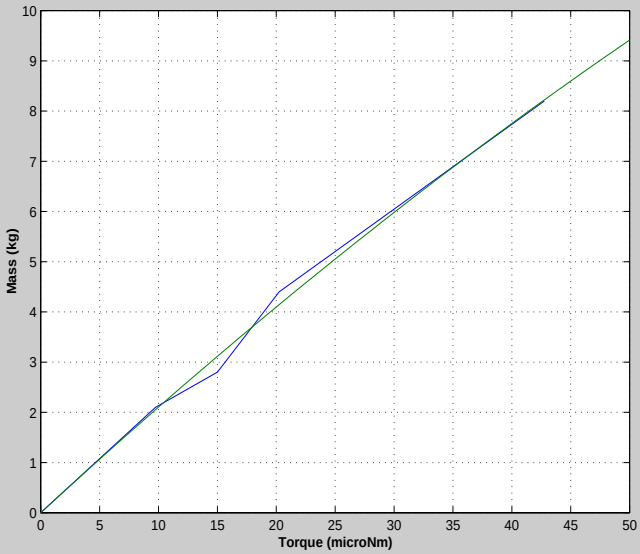


Compare magnetic torquers with thrusters at geosynchronous altitude. The specific impulse is 120 sec. and the moment arm is 0.8 m.

```
MagTComp( 120, 0.8, 75e-9 )
```

The following plot shows the trade-off between thrusters and magnetic torquers for geosynchronous orbit where the magnetic field is 75 nano-Tesla.

Figure 4-3 MagTComp



The specific impulse assumes very short pulses.

Control

Ackerman's pole placement formula can be used to critically damp a double integrator Ackerman's formula should be used with caution for higher order systems

```
a = [ 0,1;0,0];
b = [ 0;1];
p = sqrt(2)/2*[ 1+j,1-j];
k = Acker(a,b,p)
k =
   1.00000000000000  -1.41421356237310

eig(a-b*k)

ans =
  0.70710678118655 + 0.70710678118655i
  0.70710678118655 - 0.70710678118655i
```

In the following example, we use a lead/lag compensator to compensate a double integrator The double integrator plant is as follows.

```
a   = [ 0  1;0  0];
b   = [ 0;1];
c   = [ 1  0];
```

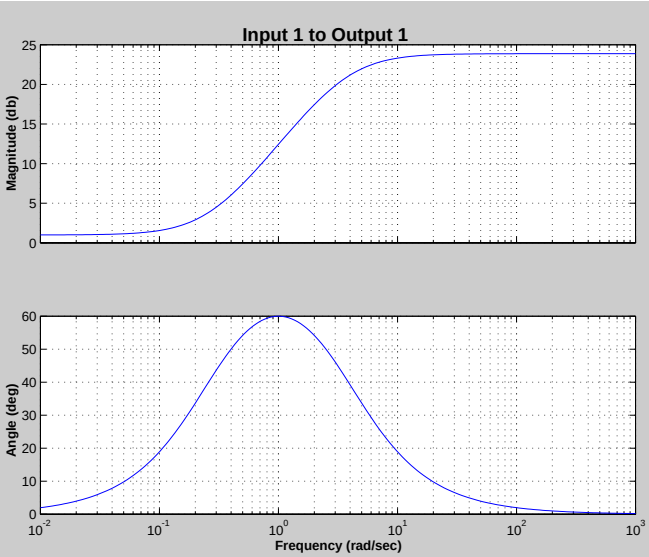
Create a lead network with 60 deg of phase lead at 1 rad/sec.

```
[ aC, bC, cC, dC ] = LeadLag( 1, 60, 1 );
```

The following demonstrates automatic plotting

```
LeadLag( 1, 60, 1 );
```

Figure 4-4 LeadLag



Generate the closed loop system

```
[ aT, bT, cT, dT] = CLoopS( a, b, c, aC, bC, cC, dC );
```

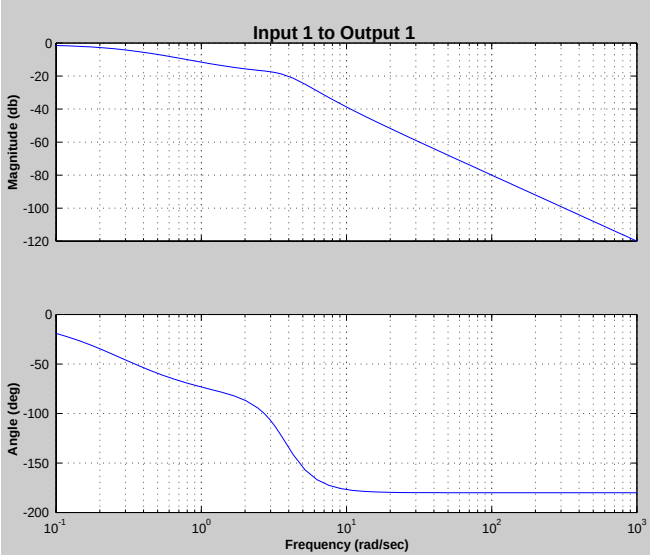
The closed loop eigenvalues

```
eig(aT)

 ans =
-1.72303339062007 + 3.41662654402684i
-1.72303339062007 - 3.41662654402684i
-0.28598402632875
```

The torque transmission is found from the closed loop system.

```
FResp(aT, bT, cT, dT)
```

Figure 4-5 Torque transmission



Create a frequency array for the following examples.

```
w = logspace(-1,1);
```

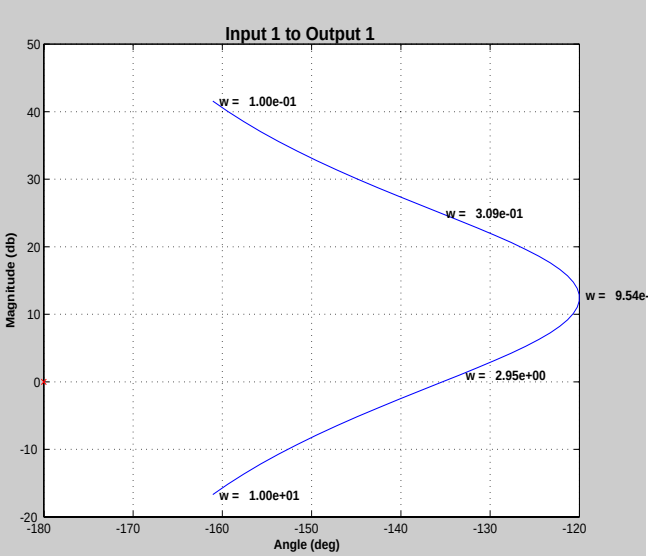
You need the open-loop plant for stability analyses.

```
[aS,bS,cS,dS] = Series(a,b,c,0,aC,bC,cC,dC);
```

Compare the Bode and Nichols plots

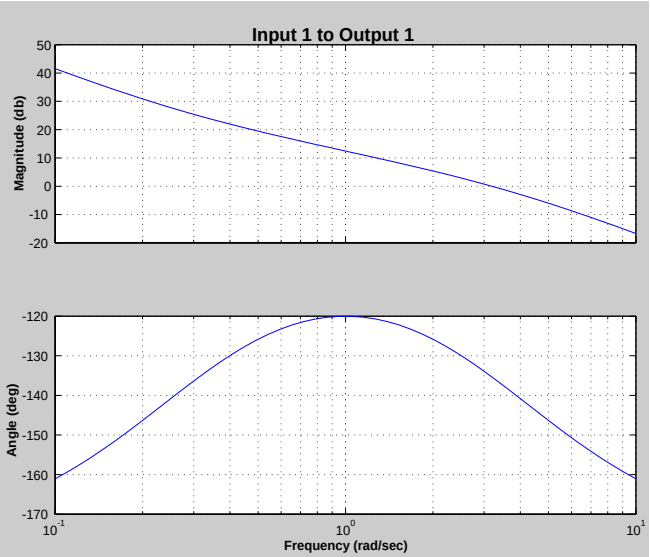
```
FResp(aS, bS, cS, dS, 1, 1, w, [], 'nichols')
```

Figure 4-6 Nichols plot




```
FResp(aS, bS, cS, dS, 1, 1, w, [])
```

Figure 4-7 Bode Plot



To compute the gain margins you first need to generate a frequency response array using FResp.

```
g = FResp(aS,bS,cS,dS,1,1,w);
```

Pass the array and frequency vector to GPMargin.

```
[ gM, wG, pM, wP] = GPMargin( g, w )

gM =
     []
wG =
     []
pM =
  46.50814520467711
wP =
   2.94705170255181
```

Coord

Euler angles are useful for initializing the attitude but are not suitable for numerical integration or simulation. Consequently we convert the Euler sequence into a quaternion.

```
q = Eul2Q([ pi/4;0;0] )

q =
   0.92387953251129
  -0.38268343236509
                  0
                  0
```

Quaternions and matrices are interchangeable. It takes fewer arithmetic operations to transform a vector using a matrix than it does for a quaternion

```
m = Q2Mat( q )

m =
   1.000000        0          0
          0        0.707107   0.707107
          0       -0.707107   0.707107
```

Dynamics

This function can return either a state-space system or the state derivative vector. MBModel has analytical expressions for the state space system

```
[ a, b, c, d] = MBModel( diag([  10000 2000
10000]),[ 0;400;0],[ 0;7.291e-5;0]);
eig(a)
```

The resulting plant has a double integrator for pitch. Roll/yaw has two pairs of complex eigenvalues, one at orbit rate (due to pointing at the earth) and one at the nutation frequency. Both modes are undamped.

```
ans =
                  0 + 0.000072910000000i
                  0 - 0.000072910000000i
                  0 + 0.039941672000000i
                  0 - 0.039941672000000i
                  0
                  0
```

Environs

Simple atmospheric models are available to compute the atmospheric density as a function of height. The first model is simpler, and much faster than the second. It also has data on some other planets.

```
AtmDens1
AtmDens2
```

Figure 4-8 AtmDens1

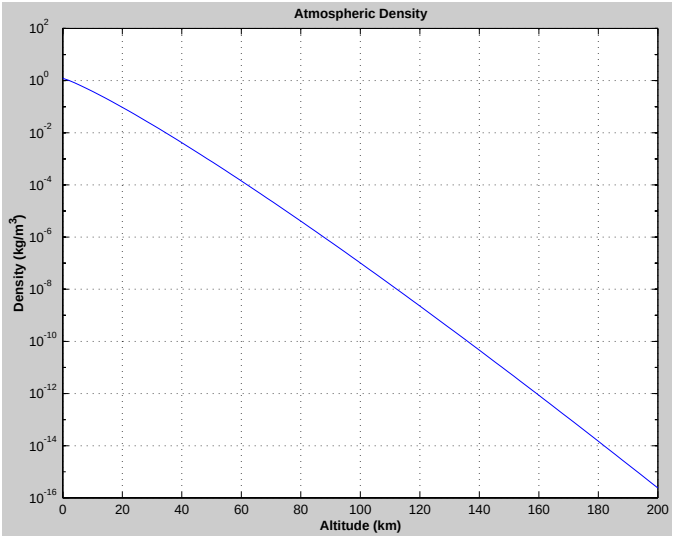
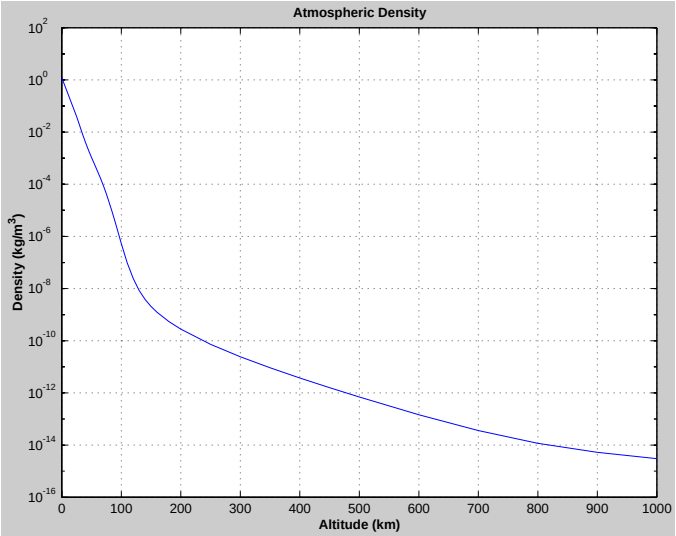


Figure 4-9 AtmDens2



Ephem

The ephemeris directory has functions that compute the time and the orientation of the earth and planets. The most widely used function is to convert calendar date to Julian Days.

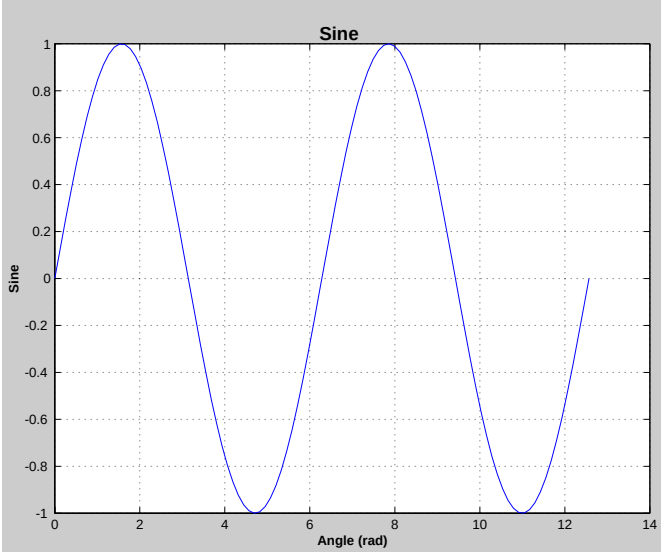
```
jD  = Date2JD
JD2Date(jD)

jD =
     2.451251504154273e+06
ans =
   [ 1999   3   14 0 0.005 0.0589292138814]
```

Graphics

Plot2D is used to plot any data. It simplifies you scripts by making most popular plotting options available through a single function. Plot2D will print out a single answer if the inputs are scalar.

```
angle = linspace(0,4*pi);
Plot2D(angle,sin(angle),'Angle (rad)','Sine',Sine)
```

Figure 4-10 A sine wave using Plot2D



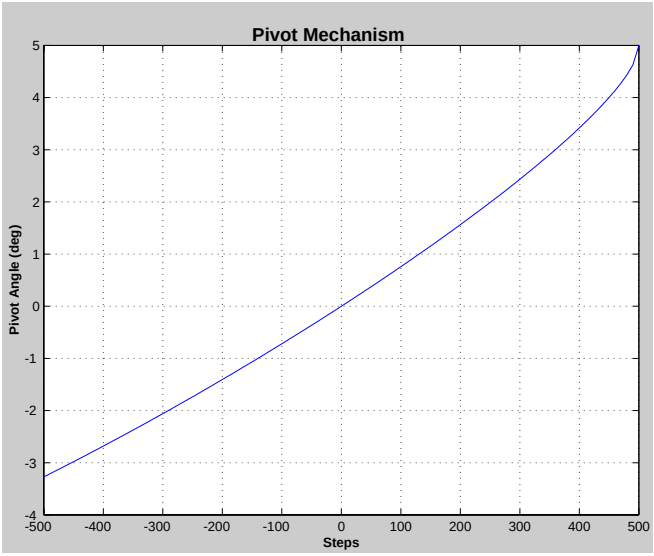
Hardware

Pivot mechanisms are nonlinear for large angles. If you type

```
PivotMch
```

you get a plot of a pivot mechanism angle versus pivot steps.

Figure 4-11 Pivot Mechanism

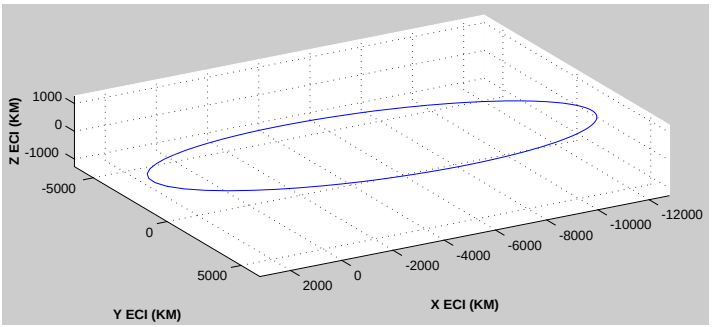


Orbit

For attitude control purposes an orbit computed by Kepler's equation is usually sufficient. `RVOrbGen` uses Kepler's equation.

```
el = [ 8000,.2,0,0,.6,0];
RVOrbGen( el  )
```

Figure 4-12 Elliptical orbit



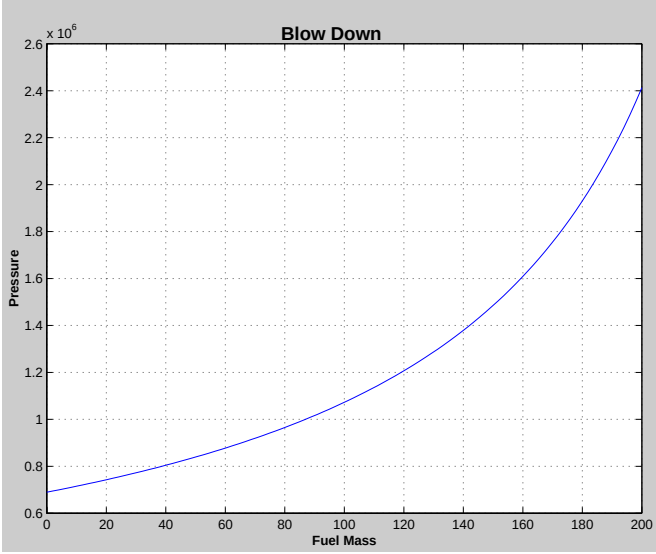

```
[ r,v] = EL2RV( el )

r =
        3200
           0
           0
v =
                    0
  13.83596536017765
   2.80468902945837
```

Prop

Many propulsion systems are blowdown designs that do not use a pressure regulator. Therefore, to know the thrust you need to know the properties of the propulsion system. The following function designs a blow down propulsion propellant system.

```
[ v, mP] = BDDesign( 200, 1000, 300, 350*6895,
100*6895 );
BloDown( mP, 1000, v, [], 300, linspace(0,200) )
```

Figure 4-13 Blowdown curve

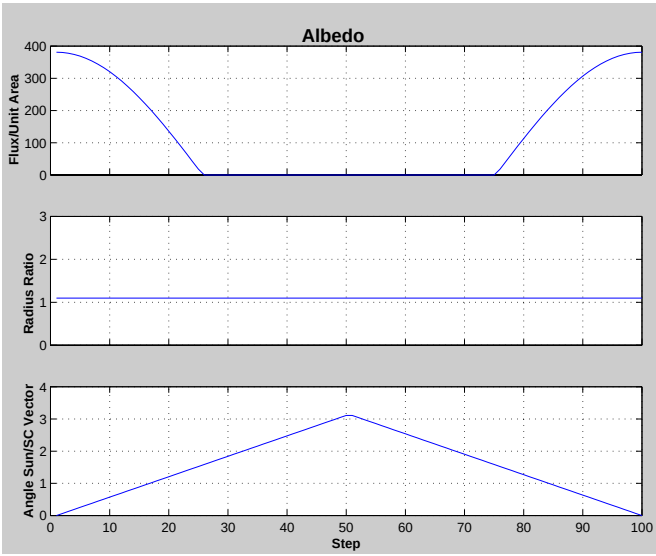


Thermal

The albedo of the earth is a significant disturbance on a spacecraft. In this model the albedo is uniform.

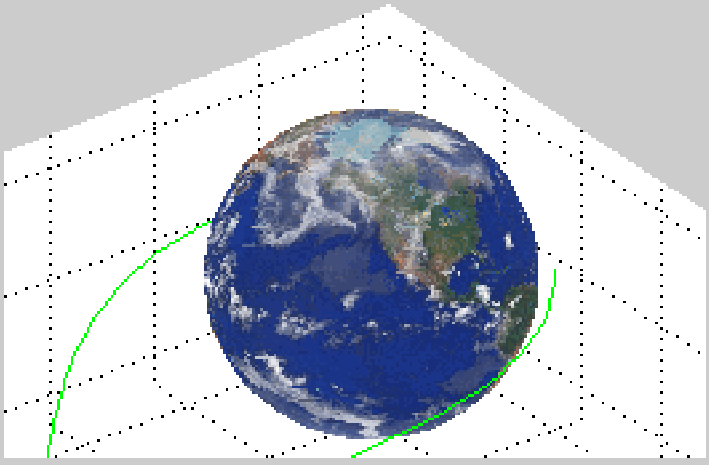
```
angle = linspace(0,2*pi);
u     = [cos(angle);sin(angle);zeros(1,100)];
r     = (7000/6378)*ones(1,100);
EarthAlb(r,u,[1;0;0])
```

Figure 4-14 Earth Albedo



CHAPTER 5

Data Structures



MATLAB 5.x have a number of useful new data types. These are used extensively in the newer functions in the toolbox. All of the CAD and GUI functions use them.

This chapter discusses data structures and cell arrays and gives some tips on how to use them.

Data Structures

Data structures allow you to collect disparate data elements into a single variable. For example, suppose you needed to pass the name of a sensor and its unit boresight vector to a function. You might write

```
u = [ 1;0;0];
name = 'Sun Sensor'
x = Sensor( name, u );
```

With data structures you can write

```
a = struct( 'u', [ 1;0;0],'name','Sun Sensor' )
```

or

```
a.u = [ 1;0;0];
a.name = 'Sun Sensor'
```

instead. Now your function call is

```
x = Sensor( a );
```

You can imagine how much more convenient passing a data structure is than passing a long list of inputs.

If you say

```
b = a;
```

`b` will be

```
b.u = [ 1;0;0];
b.name = 'Sun Sensor'
```

You cannot add data structures or use arithmetic operations on them.

You can have an array of data structures. For example:

u(1).a = eye(3);
u(1).b = rand(3,4)
u(2).a = 6*eye(3);
u(2).b = rand(3,4);

Cell Arrays

A cell array is an array in which any element can contain any other type of data structure. For example, you could implement the above data structure with a cell array

```
b{1} = [1;0;0];
b{2} = 'Sun Sensor';
```

Unlike data structures, you can concatenate cell arrays. The following

```
c = {a{:} b{:}}
```

would give you

```
c{1} = [1;0;0];
c{2} = 'Sun Sensor';
c{3} = [1;0;0];
c{4} = 'Sun Sensor';
```

You can perform operations on cell contents. For example

```
a{1} = diag([1 2 3]);
b{1} = [1;1;1];
```

when multiplied together give

```
[1;2;3];
```

Cell arrays are a convenient way of storing strings. For example you could write

```
a = ['First String     ';'Second Long String'];
```

being careful to make sure they were the same length, or write

```
a{1} = 'First String' ; a{2} = 'Second String' ;
```

`uicontrol` functions will often take cell arrays of strings.

Classes

Classes are a form of data structure in which both the data and the operations that can be done on the data are defined together. The Spacecraft Control Toolbox includes several classes. For example you might define a class called names. You could then create the method "+" which would overload the MATLAB "+" function so that if you let

```
a = names("Emily");
b = names("Stuart");
```

then

```
c = a + b;
```

would be the same as

```
c = names("Emily Stuart")
```

names would be the class constructor and "+" is a method that overloads "+".

An important aspect of a class is that you cannot get access to the internal data structure from outside of a class method. This allows the class designer control access to the data so that the user cannot use it in an incorrect manner. Object oriented design terms are listed in the following table.

Table 5-1 Object oriented design terms

Term	Definition
class	A data structure definition and functions that operate on that data structure.
constructor	A method that creates an object of type class.
instance	A variable of type class. In MATLAB if you type x = 2 you create an instance of class double.
method	A function that is part of a class.

Table 5-1 Object oriented design terms

Term	Definition
object	An instance of a class. When you type x = 2, you create an object of class double.
overloading	Giving a meaning to an operation that is specific to a class. For example, in the statespace class + means parallel connection of state space systems.
polymorphism	When a function behaves differently for different types of inputs.

SCT Classes

The following table lists all of the classes used in the toolbox and some of their methods.

Table 5-2 Classes

Class	Use	Methods
`quat`	Quaternions	matrix (converts to a transformation matrix) * quat (constructor) scalar vector ‘
`transmat`	Transformation Matrices	* quaternion (converts to a quaternion) transmat (constructor) ‘
`statespace`	State Space System	* (series connection of two systems) + (parallel connection of two systems) & (append two systems) statespace (constructor)

The operations in the table use underlying toolbox functions. For example the * in the quaternion methods uses `QMult.m`. Using the methods you can also multiply transmat objects times quat objects without converting their format yourself.

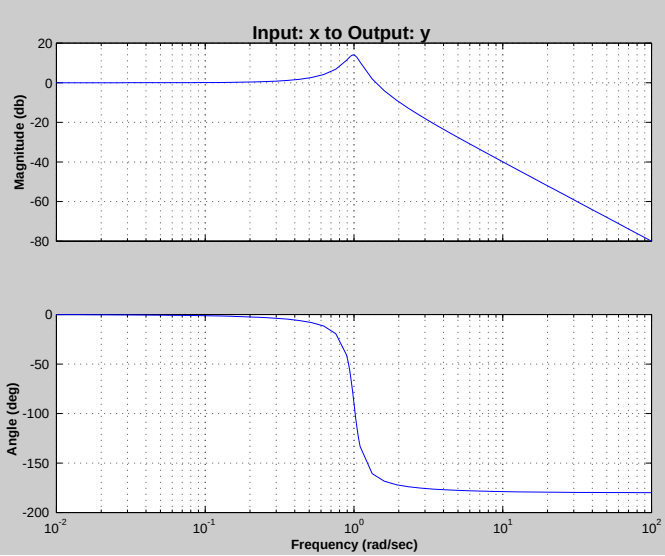
The statespace class is used in many of the new control functions. An example is shown in the following code listing.

Listing 5-1 Example of using the statespace class

```
a = [ 0 1;-1 -.2];
b = [ 0;1];
c = [ 1 0];
d = 0;
g = statespace(a,b,c,d,'Second Order',[ 'x1';'x2'],'x','y');
FrespG(g)
```

The resulting plot is shown in the following figure.

Figure 5-1 Second order response



Notice that the input and output names are in the title. The statespace class allows you to keep track of all the matrices, state, input and output names and the type of state space system in one convenient variable.

If you type

```
statespace
```

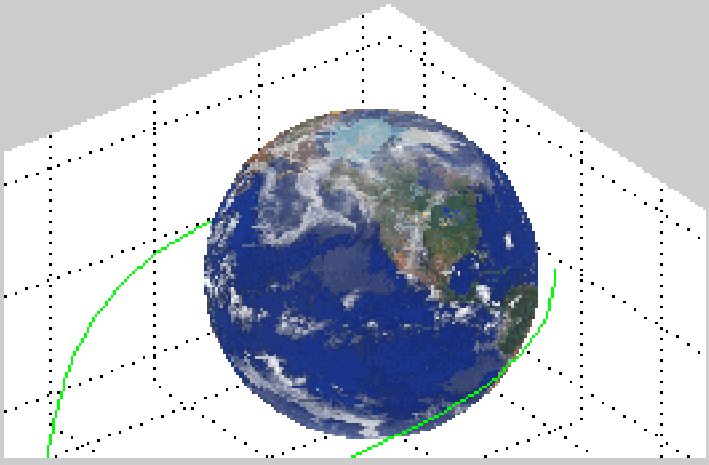
without any arguments you get

```
ans =

statespace object: 1-by-1
```

which is an empty object.

CHAPTER 6

Coordinate Transformations



This chapter shows you how to use SCT functions for coordinate transformations.

Transformation Matrices

Transforming u from its representation in A to its representation in B is easily done with a transformation matrix:

```
m = [  cos(theta),sin(theta),0;...
      -sin(theta),cos(theta),0;...
       0,0,1];
uB = m*uA;
```

This neatly ties together transformation matrices and Euler angles.

Quaternions

A quaternion is a four parameter set that embodies the concept that any set of rotations can be represented by a single axis of rotation and an angle. In this case the axis of rotation is `[ 0 0 1]'` and the angle is `theta`. Of course, the axis of rotation could also be `[ 0 0 -1]'` and the angle `-theta`. Quaternion transformations are implemented by the functions `QForm` and `QTForm`. `QForm` rotates a vector in the direction of the quaternion, and `QTForm` rotates it in the opposite direction. In this case

```
q  = Mat2Q(m);
uB = QForm(q,uA);

uA = QTForm(q,uB);
```

We could also get `q` by typing

```
q = Eul2Q([ 0;0;theta]);
```

Much as you can concatenate coordinate transformation matrices, you can also multiply quaternions. if `qAToB` transforms from A to B and `qBToC` transforms from B to C then

```
qAToC = QMult(qAToB,qBToC);
```

The transpose of a quaternion is just

```
qCToA = QPose(qAToC);
```

You can extract Euler angles by

```
eAToC = Q2Eul(qAToC);
```

or matrices by

```
mAToC = Q2Mat(qAToC);
```

If we convert the three Euler angles to a quaternion

```
qIToB = Eul2Q(e);
```

`qIToB` will transform vectors represented in I to vectors represented in B. This quaternion will be the transpose of the quaternion that rotates frame B from its initial orientation to its final orientation or

```
qIToB = QPose(qBInitialToBFinal);
```

Given a vector of small angles `eSmall` that rotate from vectors from frame A to B, the transformation from A to B is

```
uB = (eye(3)-SkewSymm(eSmall))*uA;
```

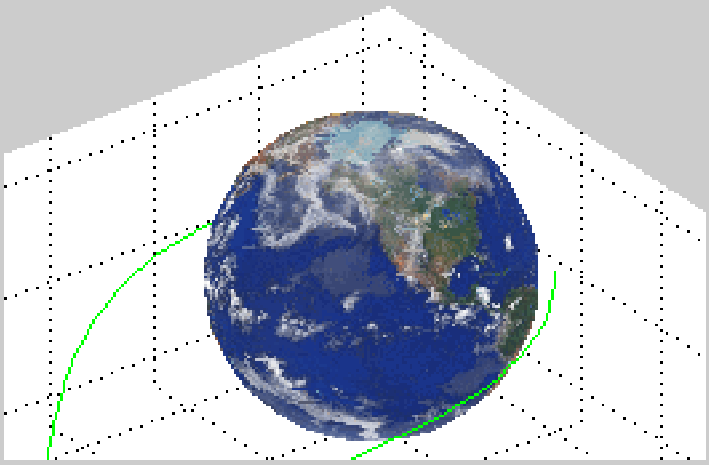
where

```
SkewSymm([ 1;2;3] )

ans =

[ 0 -3  2;
  3  0 -1;
 -2  1  0]
```

Note that `SkewSymm(x)*y` is the same as `Cross(x,y)`.

CHAPTER 7

Using the CAD Functions



This chapter shows you how to use the CAD functions. These functions allow you to work with 3-dimensional representations of your spacecraft. You can assemble spacecraft from components and use the resulting models for multi-body simulations, disturbance analysis and visualization.

Introduction

The Spacecraft Control Toolbox provides several CAD functions to simplify your design and simulation tasks. You can create a spacecraft model with `Build3DSC`, or import it using the LoadCAD function. You can interactively draw the spacecraft using DrawSC and visualize its attitude and orbit motion by passing the kinematic and position vectors to DrawSC. You can generate disturbances using Disturbance and create topological tree models using Tree-BldCAD.

Building a Spacecraft Model

Introduction

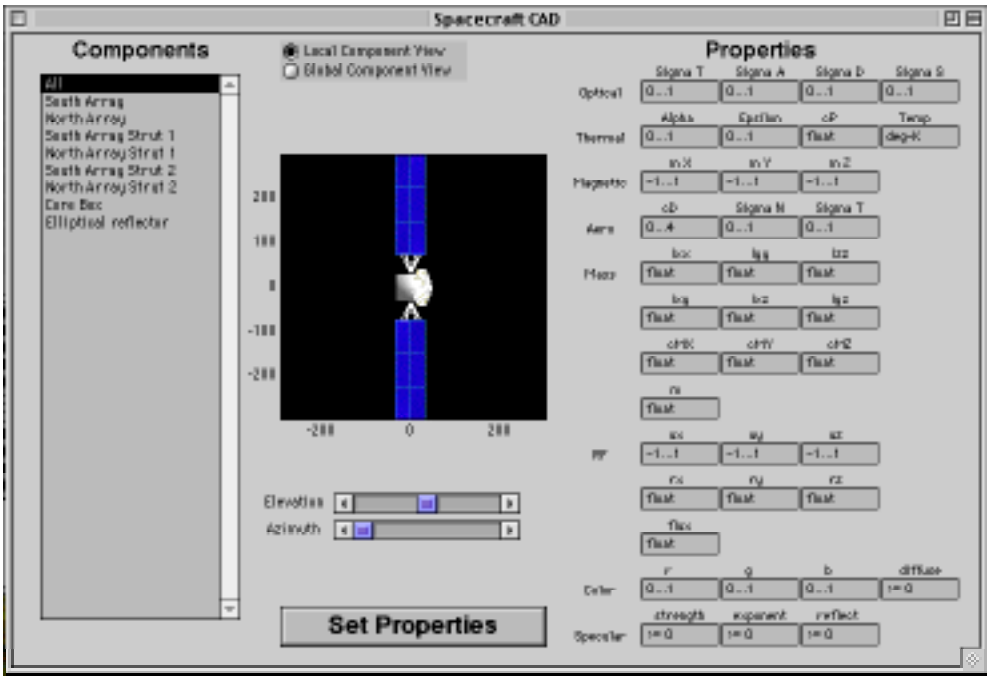
`Build3DSC` is used to build a spacecraft model. The scripts `BuildSC` and `BuildMAP` give examples. `Build3DSC` has several actions which are described in the following table. Input data is always entered in the data struc-

Table 7-1 `Build3DSC` actions

Mode	Function
`g = Build3DSC( 'init' )`	Initialize the display.
`g = Build3DSC( 'add', m )`	Add a component.
`g = Build3DSC( 'add body', m )`	Add a body which is made up of components.
`g = Build3DSC('compute paths')`	Compute paths to the core body for use in `DrawSC`

ture, m. You can enter components as vertex/face lists or as objects, such as an ellipsoid, antenna, rwa, etc. The interactive display is shown below.

Figure 7-1 Build3DSC



Each component is shown in the list on the left. If you click on the component the properties will appear on the right. You can edit them and hit Set Properties to update the properties. The valid ranges for all of the properties are shown in the properties display when All is selected. When a component is selected, the valid ranges are shown only for properties that have not been defined. You do not have to define properties for components, even if you have already defined properties for some of the components. The display in the middle shows the spacecraft. If you select Local Component View, when you select a component only that component will appear in the display. Otherwise the entire spacecraft will appear and that component will be highlighted. The Azimuth and Elevation sliders allow you to rotate the view.

Introduction

You create a spacecraft model by first entering in all of the components and then collecting them into bodies.

Adding a Component

Each component has an image associated with it. The image defines the surface geometry of the object.

You add a component with the function call

```
g = Build3DSC( 'add', m, g );
```

g is the data structure that is used to store all of the data for the spacecraft. m is the data structure that defines the data for the component.

There are two ways to define the surface geometry. One is by defining arrays of vertices and faces. The second is to specify a geometric primitive and its associated data structure. Listing 7-1 on page 4 shows how to enter a component

Listing 7-1 Adding a component by defining faces and vertices

```
% Spacecraft dimensions
%----------------------
xW = 66/2;
yW = 62/2;
zW = 74/2;

m.name = 'Nadir Panel';
m.f    = [ 1 4 3 2;1 2 3 4];
m.v    = [ -xW -yW  zW;...
           -xW  yW  zW;...
            xW  yW  zW;...
            xW -yW  zW];
m.faceColor = 'gold foil';
g           = Build3DSC( 'add', m, g );
```

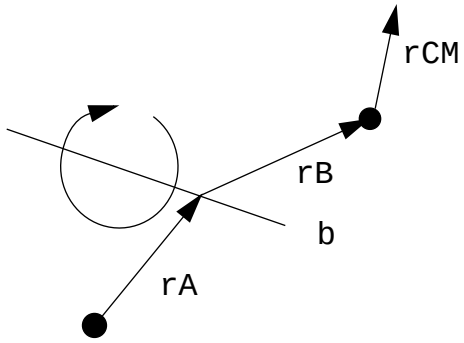
with vertices. The origin of the vertex coordinates will depend on how it is later collected into bodies. The origin is always defined as the hinge point for the body. The hinge point is the point at which the body is connected to the previous body in the chain that leads to the core. If the body is the core body, then the origin is the origin of the spacecraft.

You can modify the position and orientation of the object using the following fields

```
m.rB
m.rA
m.rCM
m.b
```

These fields modify both the picture and, when an inertia and mass is defined, effect the inertia and center-of-mass contribution of the component. These represent physical rotations of the components.

Figure 7-2 Geometry of a component



When adding a component it is sometimes desirable to show its location as it would be after it was attached to a body. This is done by specifying a position vector

```
m.rOffset
```

and a rotation vector

```
m.bOffset
```

This position vector and rotation matrix do not effect the vertices once they are attached to a body.

The color of the faces can be defined either as an RGB vector

```
m.faceColor                 = [ 1.0  0.0  0.0];
m.edgeColor                 = [ 1.0  0.0  0.0];
m.specularStrength          = 2;
m.diffuseStrength           = 0.5;
m.specularExponent          = 10;
m.specularColorReflectance  = 0;
```

for red or using a name. Unless you are very picky about the image, it is not necessary to specify anything except the faceColor.

Table 7-2 Predefined surfaces

Surfaces
'solar cell'

Table 7-2 Predefined surfaces

Surfaces
'mirror'
'gold foil'
'aluminum'
'steel'
'aluminum truss'
'radiator'
'white'
'black'

The surface properties do not have any effect on the thermal or optical properties, they are solely for display purposes.

The other way to define an object is with geometric primitives. Available geometric primitives are given in Table 7-3 on page 6

Table 7-3 Geometric primitives

m.type	m.info
'antenna'	.x x width .y y width .d depth .theta angular width of the antenna associated with d .u boresight vector .t x-axis unit vector .n number of divisions
'box'	[x y z] dimensions
'cylinder'	.rU Upper radius .rL Lower radius .zUpper Upper z limit .zLower Lower z limit

Table 7-3 Geometric primitives

m.type	m.info
'ellipsoid'	.a X radius .b Y radius .c Z radius .thetaUpper Upper angle. .n Number of divisions
'rwa'	.r radius .u spin-axis unit vector
'surface of revolution'	.a r as a function of polynomial z a(1)*z^n + a(2)*z^(n-1)... a(n)*z^0] .n Number of divisions .zUpper Upper z limit .zLower Lower z limit
'triangle'	[x y z] dimensions
'truss'	.r end points for the truss members .c connection matrix

Component properties are given in Table 7-4 on page 7.

Table 7-4 Component properties

Data structure	Component	Size	Description
.optical	.sigmaT	(1,1)	Transmission coefficient
	.sigmaA	(1,1)	Absorption coefficient
	.sigmaD	(1,1)	Diffuse reflection coefficient
	.sigmaS	(1,1)	Specular reflection coefficient
.thermal	.alpha	(1,1)	Absorptivity
	.epsilon	(1,1)	Emissivity
	.cP	(1,1)	Specific heat
	.temp	(1,1)	Temperature (deg-K)
.magnetic	.dipole	(3,1)	Dipole

Table 7-4 Component properties

Data structure	Component	Size	Description
.aero	.cD	(1,1)	Drag coefficient. If entered the next two will be ignored.
	.sigmaN	(1,1)	Normal surface accommodation coefficient
	.sigmaT	(1,1)	Tangential surface accommodation coefficient
.mass	.mass	(1,1)	Mass
	.inertia	(3,3)	Inertia matrix
	.cM	(3,1)	Center of mass wrt origin
.rF	.flux	(1,1)	flux
	.u	(3,1)	Unit vector
	.r	(3,1)	Vector from reference to rF centroid
.inside			If included the component is considered to be inside the spacecraft and will not be used in aero or optical disturbance calculations.

Adding a Body

When all components have been entered you can collect them into bodies.

Listing 7-2 Example body data structure

```
m.name         = 'Core';
m.rHinge       = [ 0;0;0];
m.bHinge       = [ 0 0 1;1 0 0;0 -1 0];
m.components   = 7:14;
m.rThruster    = zeros(3,n)
m.uThruster    = zeros(3,n)
m.magThruster  = zeros(1,n);
m.previousBody = [];
g              = Build3DSC('add body', m, g );
```

Listing 7-2 on page 9 shows an example. The data structure `bHinge` has several options

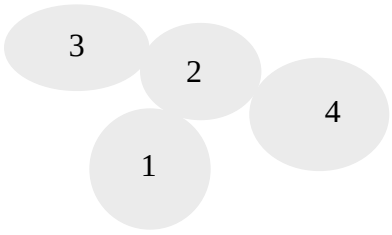
```
bHinge.b      (3,3) Transformation matrix
      .q      (4,1) Quaternion
      .a      (3,1) Unit Vector
      .angle  (1,1) Angle about the unit vector
```

You enter either `.b`, `.q` or `.a` and `.angle`. `.components` is a list of components that are attached to the body. The thruster fields define the locations of thrusters attached to the body.

Linking Bodies

The last step is to link the various bodies into a tree. An example tree is shown in Figure 7-3 on page 10. Each body is connected to the previous body by a

Figure 7-3 Topological tree



hinge. No closed loops are allowed. You create the paths by typing

```
g = Build3DSC('compute paths', g );
```

Displaying a Spacecraft Model

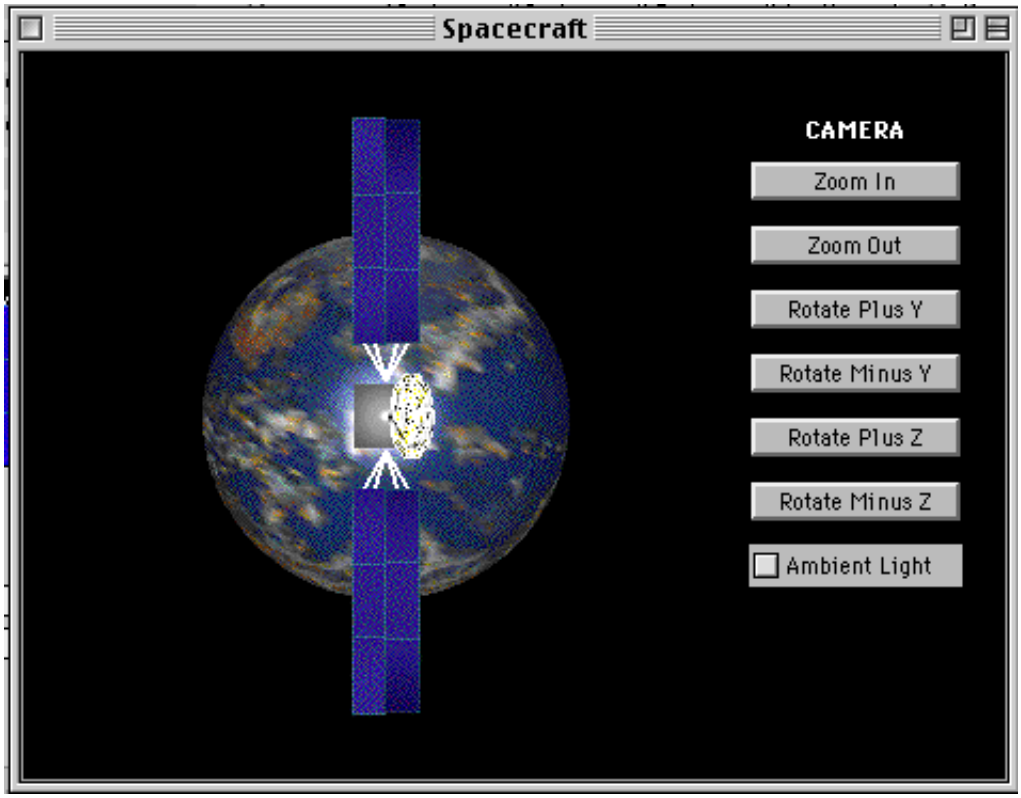
The function `DrawSC` is used to display spacecraft models. It has two actions shown in the following table. The data structure, e, contains planetary and

Table 7-5 DrawSC Modes

Mode	Function
`h = DrawSC( 'init', g, [], e )`	Initialize the display.
`h = DrawSC( 'run', g, [], e, h )`	Update the display.

spacecraft position data. g is the spacecraft data structure generated in Build3DSC. h is an internal data structure that you will not need to modify. A typical display is shown below.

Figure 7-4 DrawSC



The camera commands allow you to manipulate the point-of-view. The Ambient Light button adds artificial ambient light to the scene, which is useful for deep space missions. Planetary and moon images for all planets for which JPL has data are included.

Interfacing to the Tree Model

`TreeCAD` is a version of `TreeSim` that shows you how to create a topological tree simulation using the CAD functions. The function `TreeBldCAD` replaces TreeAdd for creating topological trees.

Computing Disturbances

Disturbances can be compute automatically using `Disturbance`. It will compute the disturbances listed in the following table. You enter the needed

Table 7-6 Disturbance

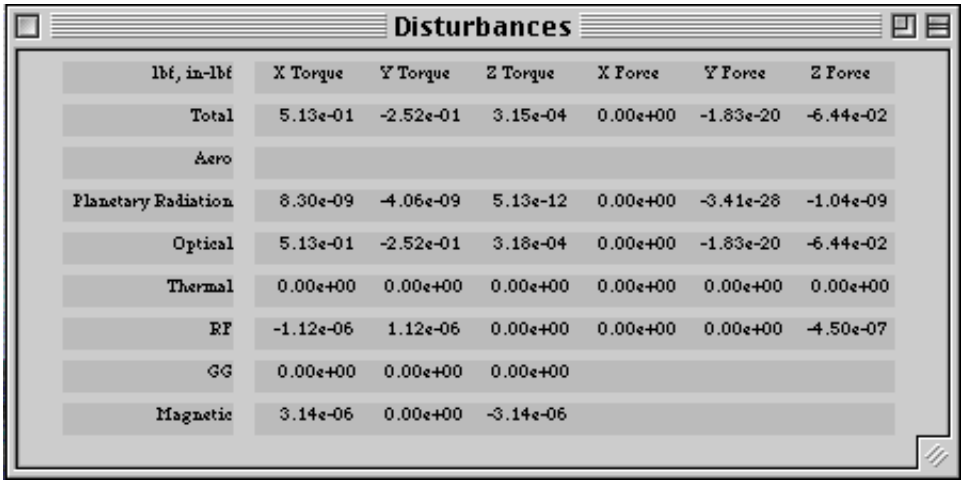
Disturbance	Description	Torque	Force
`Albedo`	Reflected sunlight from a planet	Yes	Yes
`Radiation`	Infrared radiation from a planet.	Yes	Yes
`RF`	Radiation from antennas.	Yes	Yes
`Gravity Gradient`	Torque due to the mass distribution of a satellite.	Yes	No
`Magnetic`	Magnetic dipoles on the spacecraft.	Yes	No
`Aero`	Aerodynamic interaction.	Yes	Yes
`Optical`	Sunlight.	Yes	Yes
`Thermal`	Thermal radiation from radiators, etc.	Yes	Yes

data either in your script or using the interactive property editor.

`Disturbances` can be used either in batch mode, by inputting arrays of kinematic data in the spacecraft data structure, or to calculate one disturbance vector at a time. In batch mode it will perform Fourier analysis on the disturbances torques and forces. In interactive mode you can run it concurrently with `DrawSC` to visualize the spacecraft. `SCDisturb` shows you how to run it interactively. The disturbance display, shown below, gives you a snapshot of the disturbance results.

When you enter vertex lists, you must organize the vertices into triangles, otherwise `Disturbance` will not use them.

Figure 7-5 Disturbance display



Importing CAD Models

The toolbox has the ability to import CAD files from other packages. Not all of the elements of supported packages are recognized, only the subset needed to represent 3D models. The following table lists the supported file formats and the elements that the importing functions will recognize. The script `Cassini` gives an example in which .DXF files for the Cassini orbiter and Huygens probe are loaded and displayed.

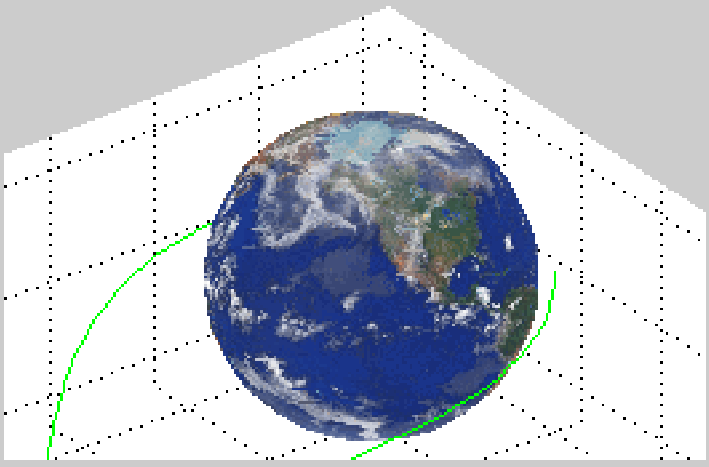
Table 7-7 CAD File Importing

Format	Applications	Supported Constructs
`3DMF`	3D World Infini-D	Mesh
`DXF`	AutoCAD	3DFACE but only triangles.
`OBJ`	Wavefront	v, f, g.

Once you import a CAD file you can use the interactive editor to add surface and material properties. Alternately, you can write a script to add the properties.

CHAPTER 8

UIElements

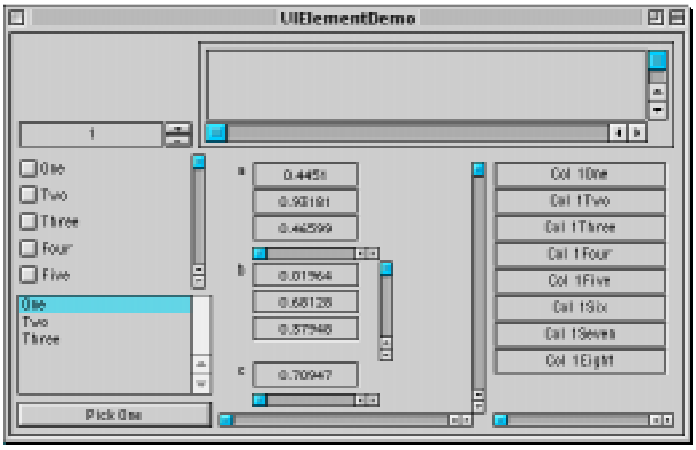


This section shows you how to use UIElements.

UIElement

`UIElement.m` is a m-file that implements a number of user interface elements that are not available in Matlab. `UIElementDemo` shows the elements that will be discussed in this section.

Figure 8-1 UIElementDemo



The UIElements shown are from left to right and top to bottom

- `EditPM`
- `EditScroll`
- `CheckBoxList`
- `DataStructureList`
- `List`
- `ListWithButton`

All, except `EditScroll`, are implemented using `UIElement`.

EditPM

EditPM is implemented as shown in Listing 8-1 on page 17. This creates an edit box with

Listing 8-1 EditPM

```
v       = {'Parent',h.fig,'Units','pixels','fontunits','pixels' };
s       = '';
h.editPM = UIElement( v{:}, 'Position',[ 5  215 130
20],'style','editpm','string','1','min',1,'max',5,'callback', s);
```

plus and minus buttons. Clicking on the buttons increments or decrements the number in the edit box. ‘min’ and ‘max’ limit the values.

EditScroll

EditScroll is implemented as shown in Listing 8-1 on page 17. This creates a scrolla-

Listing 8-2 EditScroll

```
x       = EditScroll( 'initialize', [], h.fig, [140 215 355 80])
```

ble edit window.

CheckBoxList

CheckBoxList is implemented as shown in Listing 8-1 on page 17. This is a variant on

Listing 8-3 CheckBoxList

```
s = {'One' 'Two' 'Three' 'Four' 'Five' 'Six' 'Seven' 'Eight'};
for k = 1:length(s)
  c{k,1} = {'string' s{k}};
end
h.checkBoxList = UIElement( 'parent', h.fig, 'style', 'checkboxlist',
'width', 130, 'position', [ 5 110 130 100], 'list', c );
```

the List UIElement that creates a list of checkboxes.

DataStructureList

`DataStructureList` is implemented as shown in Listing 8-1 on page 17.

Listing 8-4 DataStructureList

```
d.a = rand(3,3);
d.b = rand(5,1);
d.c = rand(1,4);
d.d = struct('x', rand(2,2), 'y', rand(3,3), 'z', 'Mike');
h.dataStructureList = UIElement( 'parent', h.fig, 'height', 40,
'position', [ 155 5 200 200], 'width', 180, 'style', 'datastructurelist',
'datastructure', d );
```

This will create a list of any data structure.

List

`List` is implemented as shown in Listing 8-1 on page 17. The list property is

Listing 8-5 List

```
s = {'One' 'Two' 'Three' 'Four' 'Five' 'Six' 'Seven' 'Eight'};
for k = 1:length(s)
  for j = 1:4
    c{k,j} = {'string' ['Col ' num2str(j) s{k}], 'style', 'edit'};
  end
end

h.list = UIElement( 'parent', h.fig, 'style', 'list', 'width', 130,...
                            'position', [360 5 135 200], 'list', c );
```

an n-by-m list of uicontrols. Any uicontrol can be included.

ListWithButton

ListWithButton is implemented as shown in Listing 8-1 on page 17. This cre-

Listing 8-6 ListWithButton

```
h.listWithButton = UIElement( 'parent', h.fig, 'style', 'listwithbutton',
'position', [ 5 5 145 100], 'buttonstring', 'Pick One' );

hS = get( h.listWithButton, 'userdata' );

set( hS.list, 'string', {'One' 'Two' 'Three'} )
```

ates a listbox with a pushbutton. Either clicking on an element of the list or hitting the pushbutton selects an element of the list. This code snippet shows you how to access the internal data structure of UIElements. In this case the code

```
hS = get( h.listWithButton, 'userdata' );
```

gives you a handle to the UIElement data structure. To set the list you type

```
set( hS.list, 'string', {'One' 'Two' 'Three'} )
```

which is setting the listbox uicontrol string.

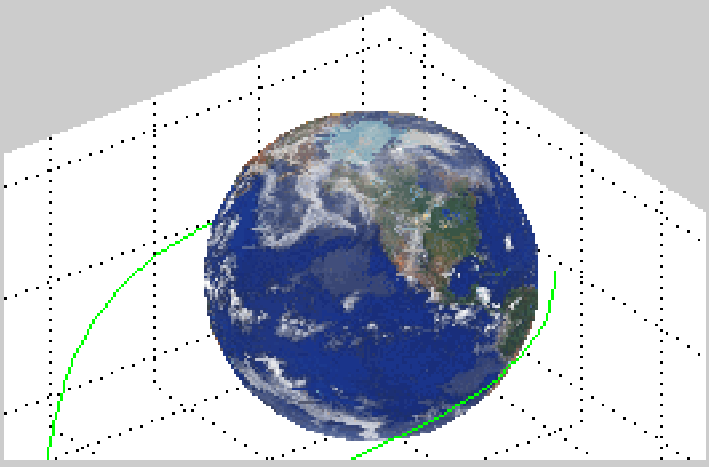
Closing UIElements

To close a `UIElement` type

```
CloseUIElement(handleToElement)
```

CHAPTER 9

GUI Plug Ins

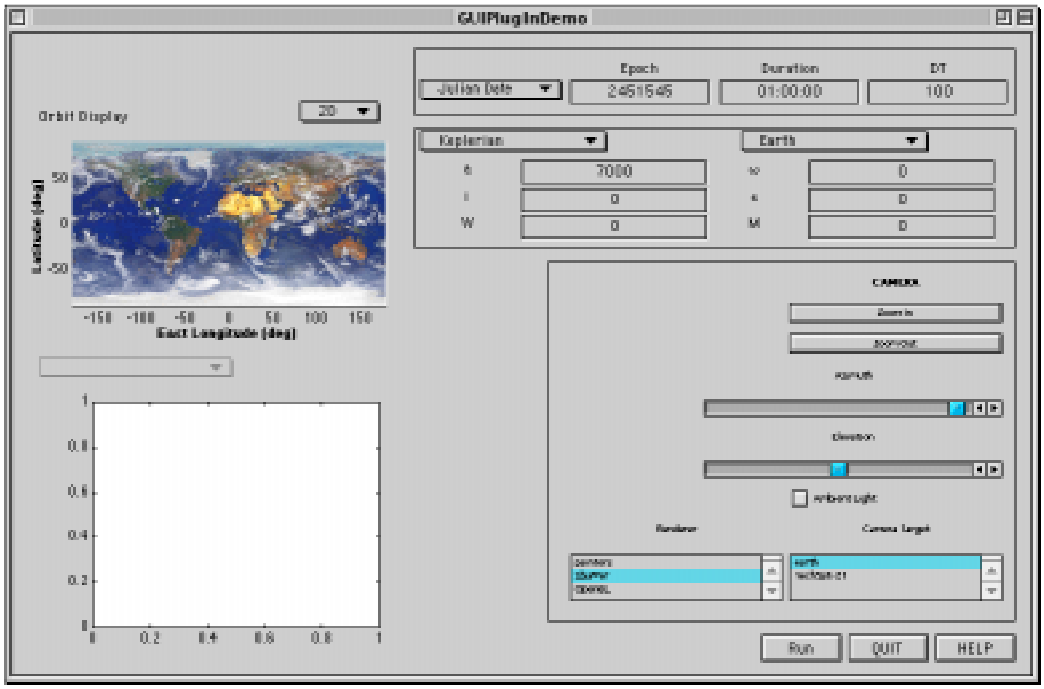


This section shows you how to use GUI Plug ins.

GUIPlugIn

`GUIPlugIn.m` is a m-file function that implements an orbit simulation using several GUI Plugin elements. This can be used as a template for your simulations.

Figure 9-1 GUIPlugIn on starting.

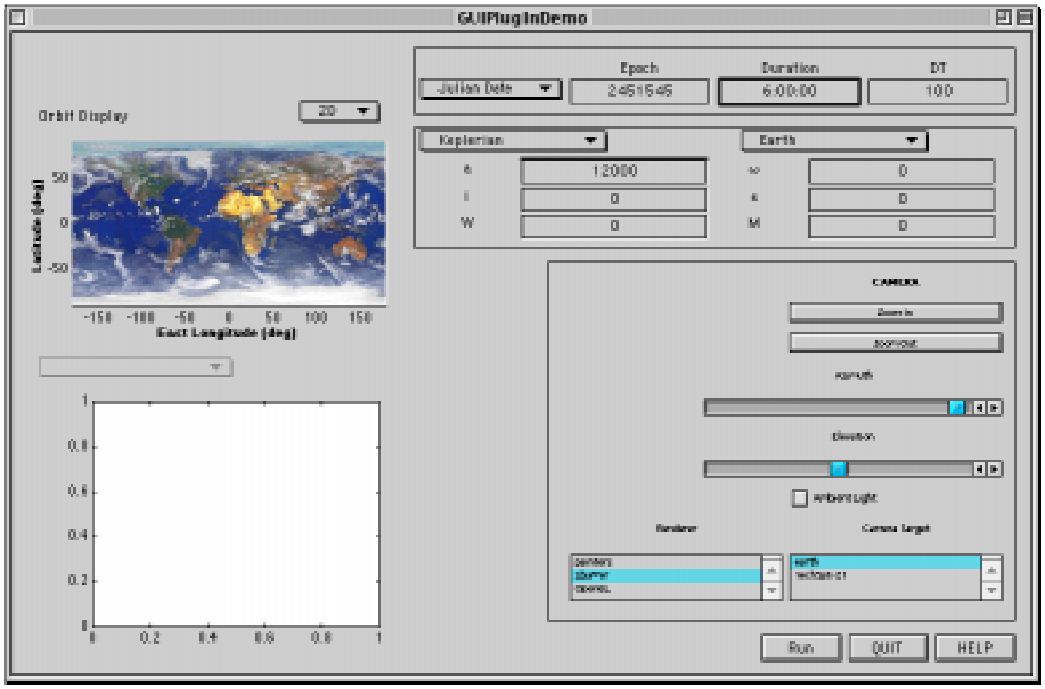


The GUIPlugins shown are from top to bottom starting on the left:

- `OrbitDisplayPlugin`
- `PlotPlugIn`
- `TimePlugIn`
- `ElementsPlugin`
- `DrawSCPlugin`

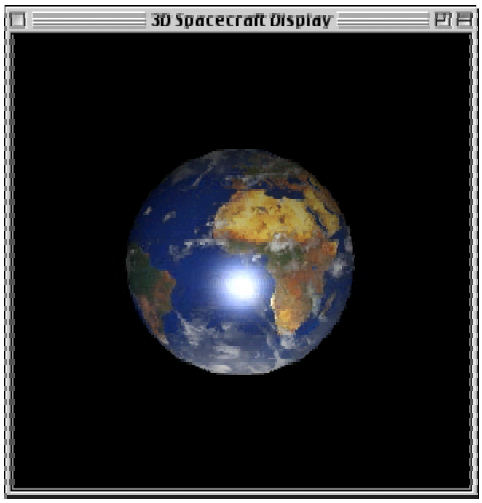
The Run, QUIT and HELP buttons are part of the GUIPlugIn function.

Figure 9-2 GUIPlugIn after changing inputs



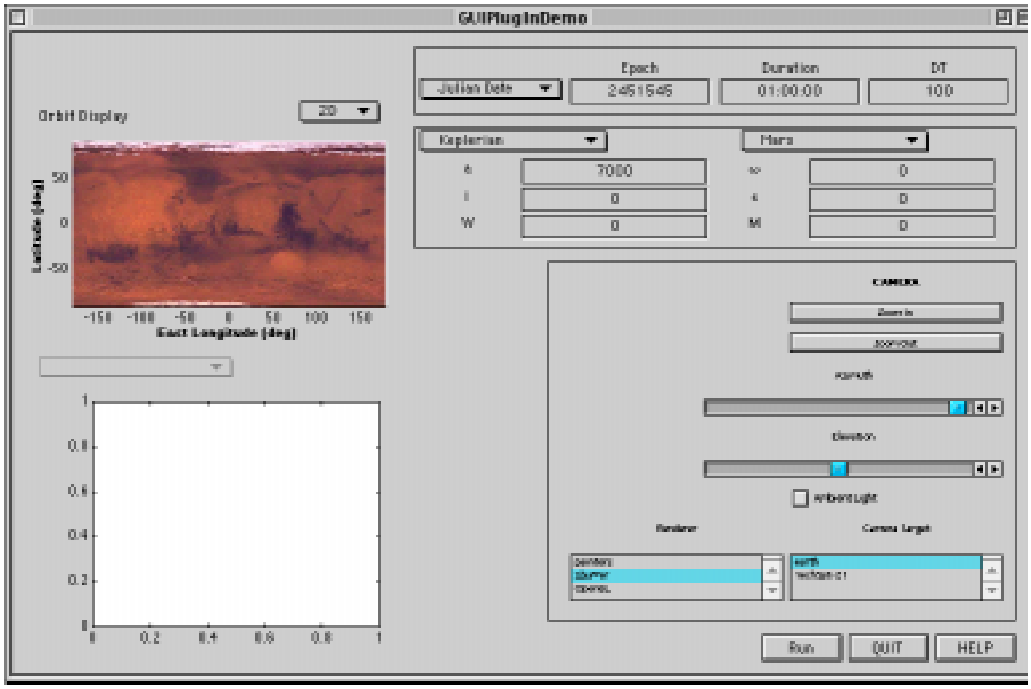
You can change any of the displayed properties in the three frames on the right. Whenever you change the properties, the changes are passed to the rest of the GUI. For example, if you select the earth as the camera center and zoom out you get the following display.

Figure 9-3 GUIPlugIn with the earth as center



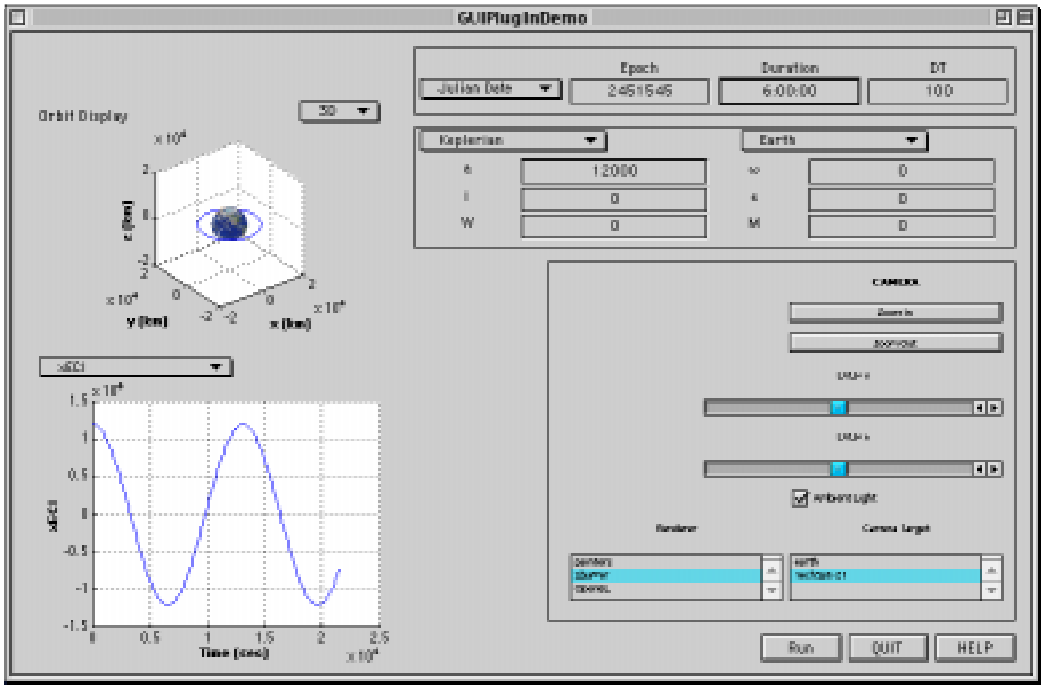
You can change planets.

Figure 9-4 Selecting planets in the elements plugin



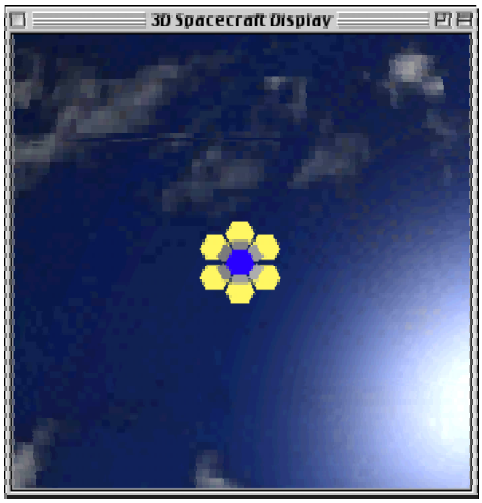
If you hit Run, the simulation will run. The results at the end are shown in the following figure.

Figure 9-5 GUIPlugIn at the end of the simulation



The 3D window (which is animated) looks like the figure below.

Figure 9-6 GUIPlugIn 3D Spacecraft Display



Writing Your Own GUI Function

`GUIPlugin` is implemented as shown in Listing 9-1 on page 26. This first

Listing 9-1 GUIPlugin

```
function GUIPlugIn( action )

% Process the input arguments
%----------------------------
if( nargin < 1 )
  action = 'create gui';
end

% Perform actions
%----------------
switch action

  case 'help'
    HelpSystem( 'initialize', 'SCHelp' );

  case 'create gui'
    h = GetH;
    if( isempty(h) )
      CreateGUI;
    else
      figure( h.fig );
    end

  case 'run'
    Run;

  case 'changed'
    Update;

  case 'quit'
    h = GetH;
    CloseFigure( h.fig );

end
```

part uses a switch statement to get the right action when `GUIPlugin` is called. All of the actions come about when you click buttons on the window. You run `GUIPlugin` by typing

```
GUIPlugin
```

If a GUIPlugin window already exists it will bring it to the front.

The `CreateGUI` subfunction draws the window and initializes all the plug

ins.

Listing 9-2 CreateGUI

```
function CreateGUI

% The figure window
%------------------
p      = [5 5 760 480];
h.fig = figure( 'name','GUIPlugInDemo','Units','pixels','Position',[40 p(4)
- 600 p(3:4)],'resize','off', 'NumberTitle', 'off','tag', 'GUI PlugIn Demo',
'CloseRequestFcn', CreateCallback( 'quit' ) );

% Buttons
%--------
v       = {'parent', h.fig, 'units', 'pixels', 'fontunits', 'pixels'};
r       = p(1) + p(3);
h.run   = uicontrol( v{:}, 'Position', [r-205  10  60 20], 'callback',
CreateCallback( 'run'  ), 'string','Run' );
h.quit  = uicontrol( v{:}, 'Position', [r-140  10  60 20], 'callback',
CreateCallback( 'quit' ), 'string','QUIT');
h.help  = uicontrol( v{:}, 'Position', [r- 75  10  60 20], 'callback',
CreateCallback( 'help' ), 'string','HELP');

% Initialize the plugins
%-----------------------
cB                    = 'GUIPlugIn( ''changed'' )';
h.orbitDisplayTag   = OrbitDisplayPlugIn( 'initialize', [], h.fig, [ 5  250
290 185], [] );
h.plotPlugInTag    = PlotPlugIn( 'initialize', [], h.fig, [ 5  10 290 235] );
h.timePlugInTag   = TimePlugIn( 'initialize', [], h.fig, [300 420 450 50], cB );
h.elementsPlugInTag   = ElementsPlugIn( 'initialize', [], h.fig, [300 320
450  90], cB );

% Initialize the 3D window
%-------------------------
sim                   = GetSimData( h );
h.g                   = load('TechSat-21');
if( isfield( sim.orbit, 'r' ) & isfield( sim.orbit, 'v' ) )
  h.g.body(1).bHinge.q = QLVLH(sim.orbit.r, sim.orbit.v );
else
  h.g.body(1).bHinge.q = [1;0;0;0];
end
h.g.name = 'TechSat-21';
if( isfield( sim.orbit, 'r' )  )
  h.g.rECI = sim.orbit.r;
else
  h.g.rECI = [sim.orbit.el(1);0;0];
end
h.g.qLVLH = h.g.body(1).bHinge.q;
h.scWindowTag = DrawSCPlugIn( 'initialize', h.g, h.fig, [400 40 350 270],
'earth', sim.time.jDEpoch );

PutH( h );
```

It first creates a figure window then creates the Run, HELP and QUIT buttons. The rest of the code initializes the plug-ins. They all have the same format. The spacecraft model is also read-in from a.mat file and used to initialize the `DrawSCPlugin`.

The following code is the orbit simulation. It gets the data from the plug-ins and updates the 3D display.

Listing 9-3 Initializing the simulation

```
function Run

% Get the simulation data
%------------------------
h       = GetH;
sim     = GetSimData( h );

% Duration
%---------
duration = datenum(sim.time.duration)*86400;

% Check duration
%---------------
if( duration == 0 )
  msgbox('The duration is zero. Will not run the simulation.');
  return
end

% Create the basic state vector from required plug-ins
%-----------------------------------------------------
x       = [sim.orbit.r; sim.orbit.v];
jD      = sim.time.jDEpoch;
nSim    = duration/sim.time.dT;
y{1}    = zeros(6,nSim);
t       = 0;

DrawSCPlugIn( 'bring to front', h.scWindowTag );
```

The simulation is run with the following code.

Listing 9-4 The simulation loop.

```
for k = 1:nSim

  % Plotting
  %---------
  y{1}(:,k) = x;
  u(k)      = t;

  % Transformation matrices
  %------------------------
  qLVLH                = QLVLH( x(1:3), x(4:6) );
  h.g.body(1).bHinge.q = QPose( qLVLH );
  h.g.rECI             = x(1:3);
  h.g.qLVLH            = qLVLH ;
  DrawSCPlugin( 'update spacecraft', h.scWindowTag, h.g, jD );

  % Propagate the orbits
  %---------------------
  x = RK4( 'FOrb', x, sim.time.dT, t, 'car', sim.orbit.mu );

  % Update the time
  %----------------
  t  =  t + sim.time.dT;
  jD = jD + sim.time.dT/86400;

end
```

The plots are created with the following code.

Listing 9-5 Plotting

```
% Plotting
%---------
if( ~isempty( y ) )
  PlotPlugIn( 'clear plots',   h.plotPlugInTag, k );
  p.xLabel = {'Time (sec)'};
  p.yLabel = {'xECI' 'yECI' 'zECI' 'vXECI' 'vYECI' 'vZECI'};
  p.title  = p.yLabel;
  PlotPlugIn( 'update labels', h.plotPlugInTag, p  );
  PlotPlugIn( 'add points', h.plotPlugInTag, struct('x', u, 'y', y ) );
  PlotPlugIn( 'plot', h.plotPlugInTag );
end

OrbitDisplayPlugIn( 'draw', h.orbitDisplayTag, {y{1}(1:3,:)},
sim.time.jDEpoch + u/86400, sim.orbit.planet );

PutH( h );
```

When you change anything in a plug in the following code is executed.

Listing 9-6 Update GUIPlugin

```
function Update

% Get the simulation data
%------------------------
h                    = GetH;
sim                  = GetSimData( h );

% Update the spacecraft state
%----------------------------
if( isfield( sim.orbit, 'r' ) & isfield( sim.orbit, 'v' )  )
  qLVLH = QLVLH(sim.orbit.r, sim.orbit.v );
else
  qLVLH = [1;0;0;0];
end
h.g.body(1).bHinge.q = QPose( qLVLH );
if( isfield( sim.orbit, 'r' )  )
  h.g.rECI              = sim.orbit.r;
else
  h.g.rECI              = [sim.orbit.el(1);0;0];
end
h.g.qLVLH            = qLVLH ;

DrawSCPlugin( 'update spacecraft', h.scWindowTag, h.g, sim.time.jDEpoch );
OrbitDisplayPlugIn( 'clear plot', h.orbitDisplayTag, sim.orbit.planet );
DrawSCPlugin( 'update planet', h.scWindowTag, sim.orbit.planet );

PutH( h );
```

This function gets the data from the elements and time plug-ins.

Listing 9-7 Get data from the plug-ins

```
function sim = GetSimData( h )

% Get the data from the plug ins
%-------------------------------
sim.orbit = ElementsPlugIn( 'get', h.elementsPlugInTag );
sim.time  = TimePlugIn    ( 'get', h.timePlugInTag     );
```

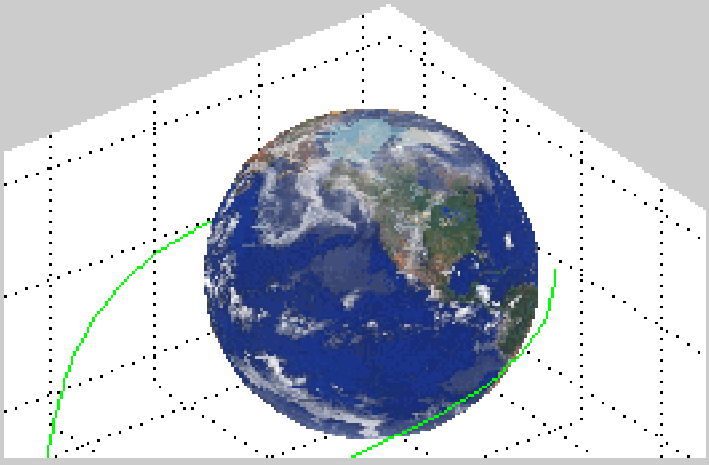
The following are utility functions for getting data from the figure handle, putting data into the figure handle and creating uicontrol callback strings.

Listing 9-8 Utilities

```
%--------------------------------------------------------------------------
%   Put the data into the figure handle
%--------------------------------------------------------------------------
function PutH( h )

set( h.fig, 'UserData', h );

%--------------------------------------------------------------------------
%   Get the data from the figure handle
%--------------------------------------------------------------------------
function h = GetH

hFig = findobj( allchild(0), 'flat', 'Tag', 'GUI PlugIn Demo' );
h = get( hFig, 'userdata' );

%--------------------------------------------------------------------------
%   Create a callback string
%--------------------------------------------------------------------------
function c = CreateCallback( action )

c = ['GUIPlugin( ''' action ''' )'];
```

CHAPTER 10

A Worked Design Example

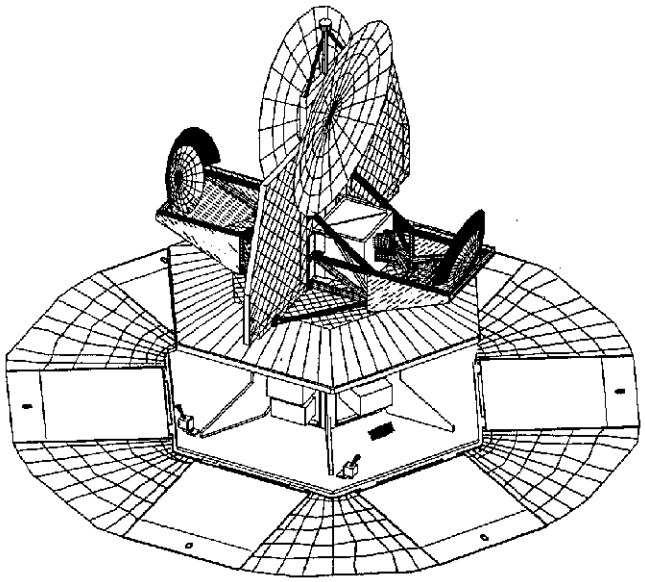


This section walks you through a complete design example, that of the Microwave Anisotropy Satellite normal and acquisition modes.

The MAP Mission

The Microwave Anisotropy Probe is a three-axis controlled spacecraft. It employs reaction wheels for nominal attitude control with

Figure 10-1 The Microwave Anisotropy Probe



star trackers and gyros for sensing. Ten hydrazine thrusters are available for backup attitude control, orbit adjust maneuvers and momentum unloading.

Control Modes

The MAP spacecraft requires three control modes:

- Mission control mode using reaction wheels with thrusters for momentum unloading. This mode would also include the acquisition function.
- Backup mission control mode with thrusters for three-axis attitude control
- Orbit adjust mode using thrusters for three-axis attitude control. The reaction wheels are kept in a tachometer mode during orbit changes.

Optionally, a safe hold mode could be added.

Sensing and Actuation

Nominally, MAP uses gyros for attitude determination and a star tracker to correct for gyro drift and for absolute attitude information. The gyro and star tracker data would be integrated using an extended iterated Kalman filter. A backup mode using just the star tracker would also be available.

During most of the mission the reaction wheels are used for attitude control. Secular momentum growth due to solar pressure would be controlled using the thrusters. Momentum unloading could be autonomous or ground commanded. During orbit adjust modes the thrusters would be used for attitude control since the disturbances due to the orbit adjust thrusters would quickly saturate the reaction wheels.

Control System Design

The control system design is a modification of Princeton Satellite Systems' standard three axis control system. The control system employs tachometer inner loops for the three reaction wheels. The outer loops are two Proportional Integral Differential (PID) loops for the transverse axes and a Proportional Integral (PI) loop for the spin axis. The integral term in the PID controllers insures accurate tracking of the spin axis target. The inner reaction wheel loops are PI controllers. The outer loops output an acceleration demand that is converted to a wheel speed demand and passed to the reaction wheel tachometer loops. The inner loops insure proper reaction wheel response regardless of bearing friction magnitude and uncertainty. A sampling rate of 4 Hz was chosen for the digital implementation of this control system.

Simulation Results

Normal mode acquisition was simulated using the `TMAPSim` script.The mass properties and the actuator characteristics for the baseline MAP spacecraft were not available so generic parameters were used in the simulation. The spacecraft model is for a gyrostat with three reaction wheels. All nonlinear terms are included. The simulation models do not include sensor noise or external disturbances.

The spacecraft is initially spinning at 0.464 rpm with the spin axis aligned with the inertial Z axis. The desired precession angle command of 22.5 deg is fed into the control loops through a low pass filter to eliminate transients. The gains of the filter are chosen so that acquisition is completed within ten minutes.

Using the Spacecraft Control Toolbox

In this section we will walk through the `TMAPSim` script. Once you understand how is was written you will be able to easily design and build your own control systems and simulations. Listing 10-1 on page 4 This is the header which

Listing 10-1 TMAPSim: File header

```
%--------------------------------------------------------------------
%  Implements and simulates the MAP normal mode control.
%--------------------------------------------------------------------
%   Copyright 1996 Princeton Satellite Systems, Inc. All rights reserved.
%--------------------------------------------------------------------

% Clean up the workspace
% ----------------------
close all

% Global for the time GUI
%------------------------
global simulationAction
simulationAction = ' ';

% Constants
% ---------
degToRad = Constant('deg to rad');
radToDeg = Constant('rad to deg');
rPMToRPS = Constant('rpm to rad/sec');
```

defines a few useful constants. The global variable is used to control the time GUI. We routinely close all windows.The sampling interval is both for the con-

Listing 10-2 TMAPSim: Sampling intervals

```
% The control sampling period and the simulation integration time step
% --------------------------------------------------------------------
tSamp       = 0.25;

% Number of sim steps
% -------------------
nSim        = 4*3600;

% Plot every nPMax steps
% ----------------------
nPMax       = 40;
nPlot       = nSim/nPMax;

% Print the time to go message every nTTGo steps
% ----------------------------------------------
nTTGo       = 1000;
```

troller and the simulation. This is fine if your controller effectively sets the highest frequencies in the simulation. If your model includes higher frequency terms (which would cause jitter) you would need to use a smaller integration steps. The rest of the quantities are used to cause the simulation plot less frequently than once per integration step..

Listing 10-3 TMAPSim: Mass properties

```
% Spacecraft Inertias
% -------------------
inr         = [2000,0,0;0,2000,0;0,0,4000];
inrRWA      = 1;
inrW        = [inrRWA,inrRWA,inrRWA];
invInr      = inv(inr);
tDist       = [0;0;0];

% Wheel spin axis unit vectors
% ---------------------------
uW          = eye(3);
```

Listing 10-3 on page 5 defines the spacecraft mass properties. This set is not based on the actual MAP properties but is for demonstration purposes only. We are using a gyrostat (rigid body + wheels) model that allows for any number of reaction wheels so we must include the spin axis inertias of the wheels. We will use three orthogonal wheels.

Listing 10-4 TMAPSim: Tachometer loop design

```
% ---------------------------------------------------------------------
% Design the control loops
% ---------------------------------------------------------------------
% RWA Tach loops
% --------------
zeta        = 0.7071;   % Damping ratio
wN          = 1.0;      % Closed loop undamped natural frequency

[aTL,bTL,cTL,dTL] = PIDesign( zeta, wN, inrW(3), tSamp, 'Delta' );
```

First design the inner tach loops for the reaction wheels. Our attitude control loops will actually command rate changes to the wheels. The inner loops will handle uncertainty in friction, wheel motor scale factors, etc. `PIDesign` uses pole placement to design a PI loop. In this example we will implement the controllers in delta form so that state space equations produce delta's to the control commands, not the actual value.

Next, design the attitude loops. We are only controlling the spin axis orientation so we only need two loops, roll and pitch (we are spinning about yaw.) `PIDMIMO` designs a PID controller with a rate filter.

Listing 10-5 TMAPSim: ACS design

```
% Attitude Loops
% --------------
zeta        = 0.7071;   % Damping ratio
wN          = 0.5;      % Closed loop undamped natural frequency
wR          = 4.0;      % Rate filter break frequency
tau         =  50;      % Integrator time constant

[aRoll ,bRoll, cRoll, dRoll]  = PIDMIMO( inr(1,1), zeta, wN, tau, wR,
tSamp, 'Delta');
[aPitch,bPitch,cPitch,dPitch] = PIDMIMO( inr(2,2), zeta, wN, tau, wR,
tSamp, 'Delta');
```

Listing 10-6 on page 6 gives the yaw rate loop that will maintain the MAP spin rate. This uses the same PI controller as the reaction wheels but with a slower time constant.

Listing 10-6 TMAPSim: Rate loop

```
% Rate Loops
% ----------
zeta        = 1.0;   % Damping ratio
wN          = 0.5;  % Closed loop undamped natural frequency

[aYaw,  bYaw,  cYaw,  dYaw]  = PIDesign( zeta, wN, inr(3,3), tSamp,
'Delta' );
```

Now that the controller design is complete it is time to run a simulation to test the system. Of course we should also do frequency domain analysis of the entire controller. Even though we used SISO design techniques this is really a MIMO design. The controller has fast inner loops to control the wheels and slower outer loops to control the attitude. Several tools exist in SCT for analyzing such systems.

The first step is to initialize the control system. We've turned off the momentum management system.

Listing 10-7 TMAPSim: Initialize the control system

```
% Initialize the control system
% ----------------------------
xTL             = zeros(3,1);
xRoll           = [0;0];
xPitch          = [0;0];
xYaw            = 0;
tC              = [0;0;0];
spinRate        = 0.464*rPMToRPS;
precRate        = 2*pi/3600;
precAngleDemand = 22.5;
precAngle       = 0;
precAngleGain   = 0.995;

% Momentum Management gain
% ------------------------
kMM         = 0.000;

% The control distribution matrix converts
% torque demand to angular acceleration demand
% ------------------------------------------
aRWA       = eye(3)/inrW(3);
wRWAC      = [0;0;0];
```

The following code sets up the simulation. We allocate memory for plotting arrays to save processing time. We also set up the time statistics function which will display time statistics.

Listing 10-8 TMAPSim: Set up the simulation

```
% Plotting arrays
% ---------------
cPlot      = zeros( 3,nPlot);
hPlot      = zeros( 3,nPlot);
tPlot      = zeros( 1,nPlot);
xPlot      = zeros(10,nPlot);
zPlot      = zeros( 3,nPlot);
rPlot      = zeros( 3,nPlot);
pPlot      = zeros( 2,nPlot);
ePlot      = zeros( 2,nPlot);
wPlot      = zeros( 3,nPlot);
yPlot      = zeros( 3,nPlot);

% Time statistics function
% ------------------------
tToGoMem      = zeros(1,3);
ratioRealTime = 0;

% Initial conditions
% ------------------
%                 q         w            wRWA
x          = [[1;0;0;0];[0;0;spinRate];[0;0;0]];

dTSim      = tSamp;
t          = 0;
nP         = 0;
kP         = 0;
tW         = zeros(3,1);
roll       = 0;
pitch      = 0;
yaw        = 0;

% Initialize the time display
%----------------------------
tToGoMem.lastJD        = 0;
tToGoMem.lastStepsDone = 0;
tToGoMem.kAve          = 0;
ratioRealTime          = 0;
[ ratioRealTime, tToGoMem ] =  TimeGUI( nSim, 0, tToGoMem, 0, tSamp,
'TMAPSim' );
```

This is the start of the simulation loop. The first function displays the status message in the command window.

Listing 10-9 TMAPSim: Start of the simulation loop

```
% Run the simulation
% ------------------
for k = 1:nSim

  % Display the status message
  %--------------------------
  [ ratioRealTime, tToGoMem ] = TimeGUI( nSim, k, tToGoMem, ratioRealTime,
tSamp );
```

We are using very simple sensor models for this demonstration. Just the true states without sensor dynamics, scale factor errors, noise or any other effects that are quite important in practice.

Listing 10-10 TMAPSim: Sensor models

```
% ------------------------------------------------------------------------
% Sensors
%-------------------------------------------------------------------------
% RWA Tachometer
% --------------
  wTach  = x(8:10);

% Gyros
% -----
  wCore  = x(5:7);

% Attitude
%---------
  qIToB  = x(1:4);
```

Finally, we get to the control system implementation. The first part is the momentum management system which is just a proportional controller.

Listing 10-11 TMAPSim: Control system

```
% ----------------------------------------------------------------------
% The Attitude Control System
%----------------------------------------------------------------------
% Momentum Management
% Neglect the rate errors in the body component-assume exact tracking
% ---------------------------------------------------------------
  hTotal = QTForm( qIToB, inr*[wCore(1);wCore(2);wCore(3)] +
inrRWA*uW*(wTach+wCore) );

% Proportional controller for momentum
% We could feedforward this to the controller
% ---------------------------------------
  tMM    = QForm( qIToB, -kMM*hTotal );
```

We next compute the errors. These are what we will be controlling. Notice that the commanded precession angle is filtered. This makes the system a two-degree-of-freedom control system and prevents the operator from putting in a command that would saturate the actuators.

Listing 10-12 TMAPSim: Error computation

```
% Compute the errors
%-------------------
  precAngle  = precAngleGain*precAngle + (1-
precAngleGain)*precAngleDemand;
  cP         = CosD(precAngle);
  sP         = SinD(precAngle);
  xT         = [sP*cos(precRate*t);sP*sin(precRate*t);cP];
  xTB        = QForm( qIToB, xT );
  rollError  =  asin(xTB(2));
  pitchError = -asin(xTB(1));
  yawError   = wCore(3) - spinRate;
```

These loops control the attitude and spin rate. The output is torque demand.

Listing 10-13 TMAPSim: Attitude loop

```
% The attitude control loops
% --------------------------
  tC(1)  = -cRoll*xRoll - dRoll*rollError;
  xRoll  =  xRoll + aRoll*xRoll + bRoll*rollError;

  tC(2)  = -cPitch*xPitch - dPitch*pitchError;
  xPitch =  xPitch + aPitch*xPitch + bPitch*pitchError;

  tC(3)  = -cYaw*xYaw - dYaw*yawError;
  xYaw   =  xYaw + aYaw*xYaw + bYaw*yawError;
```

We convert torque into a demand on the wheels which we integrate to get wheel speed demand.

Listing 10-14 TMAPSim: Compute wheel speed demand

```
% Convert torque demand to RWA angular acceleration demand
% --------------------------------------------------------
  wDRWA  = -aRWA*tC;

% Integrate to get wheel speed demand
% -----------------------------------
  wRWAC  = wRWAC + tSamp*wDRWA;
```

The wheels are commanded through the tach loops.

Listing 10-15 TMAPSim: RWA tach loops

```
% The RWA Tach Loops
% ------------------
  wError =  wTach - wRWAC;
  tRWA   = -dTL*wError - cTL*xTL;
  xTL    =  xTL + aTL*xTL + bTL*wError;
```

This is the simulation. We are using the gyrostat model for this simulation.The gyrostat model ‘FGs’ is called by RK4. RK4 is a Fourth-Order Runge-Kutta method that works well for most spacecraft control simulation problems. It does not do automatic error correction or computation. This means that you cannot tell how accurately it is integrating the equations of motion from a single run. To check on numerical errors you should run your simulation at half the time step and compare the results. If they are significantly different

you will need to go to a smaller step size. The RK45, with somewhat greater overhead, can

Listing 10-16 TMAPSim: Simulation

```
% ------------------------------------------------------------------------
% Update the equations of motion
%-------------------------------------------------------------------------
x        = RK4('FGs',x,dTSim,t,inr,invInr,tDist+tMM,inrW,uW,tRWA');
t        = t + dTSim;
```

be used in situations where you need to control the precision of your simulation.

In the following listing we are saving information for the plots. The code under the com-

Listing 10-17 TMAPSim: Saving plot information

```
% Plotting
% --------
  if nP == 0,
    kP          = kP + 1;
    xPlot(:,kP) = x;
    tPlot(1,kP) = t;
    cPlot(:,kP) = tC;
    hPlot(:,kP) = hTotal;
    zPlot(:,kP) = tMM;
    xI          = QTForm(qIToB,[0;0;1]);
    rPlot(:,kP) = xI;
    pPlot(:,kP) = acos([xI(3);xT(3)]);
    ePlot(:,kP) = [rollError;pitchError];
    wPlot(:,kP) = xT;
    yPlot(:,kP) = xTB;
    nP          = nPMax - 1;
  else
    nP          = nP - 1;
  end

  % Time control
  %-------------
  switch simulationAction
    case 'pause'
      pause
      simulationAction = ' ';
    case 'stop'
      return;
    case 'plot'
      break;
  end
end
```

ment time control allows you to control the script from the time GUI.

After the simulation is done we plot the results.

Listing 10-18 TMAPSim: Plotting

```
Plot2D(tPlot,xPlot( 1: 4,:),'Time
(sec)',['Qs';'Qx';'Qy';'Qz'],'Quaternion')
Plot2D(tPlot,xPlot( 5: 7,:),'Time (sec)',['Wx';'Wy';'Wz'],'Body Rates')
Plot2D(tPlot,xPlot( 8:10,:),'Time (sec)',['W1';'W2';'W3'],'Reaction
Wheels')
Plot2D(tPlot,cPlot,'Time (sec)',['X';'Y';'Z'],'Control Torque Demand')
Plot2D(tPlot,hPlot,'Time (sec)',['X';'Y';'Z'],'Body Momentum')
Plot2D(tPlot,zPlot,'Time (sec)',['X';'Y';'Z'],'Momentum Management
Torque')
Plot2D(tPlot,pPlot*radToDeg,'Time (sec)','Precession Angle
(deg)','Precession Angle')
PlotV([rPlot;wPlot],'X','Y','Z','Spin Axis and Target')
Plot2D(tPlot,ePlot*radToDeg,'Time (sec)',['Roll  (deg)';'Pitch
(deg)'],'Attitude Errors')
```

Figure 10-2 on page 13 shows the results of `PlotV`. It shows the spin axis unit vector. It starts as [0;0;1] (the highest point on the plot) and as the spacecraft acquires picks up y and x components until it traces out a circle in the xy-plane.

Figure 10-2 TMAPSim: Motion of the unit vector.

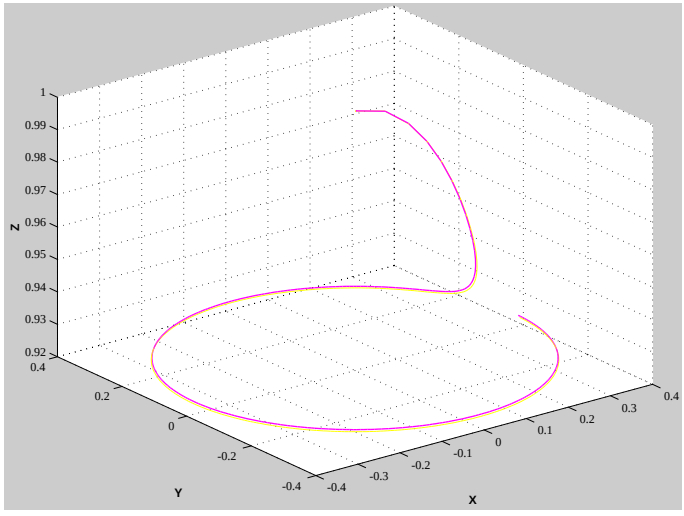
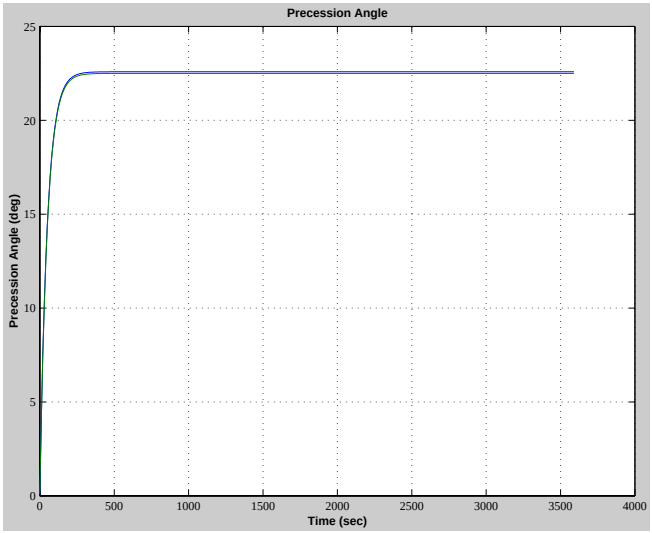


Figure 10-3 TMAPSim: Precession of the unit vector



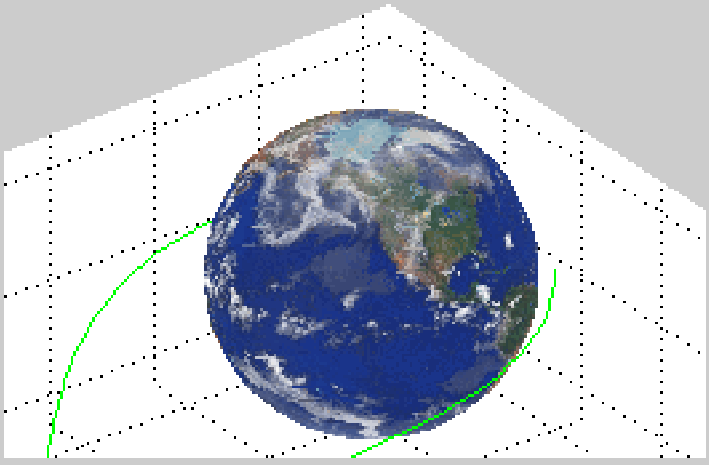
Summary

As can be seen from the plots, the control system successfully acquires and tracks the target. The overall pointing accuracy is better than 0.2 deg. All of the plots were generated using Spacecraft Control Toolbox functions. All of the design functions and simulated functions fit in a single MATLAB script. The second plot shows the unit vector for the spin-axis of the spacecraft in three dimensional space. Initially, it is aligned with the z-axis so its x and y components are zero. After acquisition the x and y-axes trace out a circle in the xy-plane and the z value is constant.

The control system presented here is only part of the overall control architecture that includes not only the other modes but also the attitude determination function, the momentum unloading system and the telemetry and command functions. Although the spacecraft is mostly single string, a fault detection and isolation system might also be included to protect the spacecraft against temporary problems.

CHAPTER 11

Simulation



Several different functions are included in the toolbox for simulating spacecraft attitude and orbit dynamics. They range from simple rigid body models to sophisticated multi-body models.

Dynamics Models

The following table lists the dynamics models and their capabilities. Some

Table 11-1 Attitude Simulation Functions

Function	State Space Model Output?	Purpose	Support Functions
ASim		Attitude simulation of a rigid body.	
DIModel		Double integrator model.	
FAKM		Solid rocket simulation.	
FDC		constrained equations in the form MxDot = F.	
FGs		Gyrostat dynamics with quaternion kinematics.	
GSModel		Gyrostat dynamics.	
FPivot		Spacecraft with pivoted momentum wheel and quaternion kinematics.	
SCwPivot		Spacecraft with pivoted momentum wheel.	
FRB		Rigid body with quaternion kinematics.	
RBModel		Rigid body.	
FTB		Two body with quaternion kinematics.	
TBModel		Two body.	
FXModel		Flexible spacecraft model.	
MBModel		Momentum bias spacecraft (dynamics only).	

Table 11-1 Attitude Simulation Functions

Function	State Space Model Output?	Purpose	Support Functions
`Tree`		Topological tree.	TreeAdd TreeH TreeInit TreePlot TreePrnt
`VIModel`		Body with time varying inertia.	
`WireFRB`		Deployable wire model	WireDMch WireH WireInit WirePlot
`WireC`		Deployable wire model employing kinematic constraints.	WireDMch WireH WireInit WirePlot

models are represented with two functions, one that has only the dynamics (such as RBModel) and the other which appends quaternion kinematics to the equations of motion. You can get linearized models of those that don't directly output state space equations by using the `Jacobian` function

A simple rigid body attitude simulation can be embodied in the function.

```
xDot = ASim( x, t, inertia, torque )
```

```
xDot = [ QIToBDot( x(1:4), x(5:7)); RBModel( x(5:7),
inertia, torque)];
```

The first four elements of x form the quaternion from the inertial frame to the body frame. The last three elements of x are the inertial body rates measured in the body frame. t is not used in `ASim`. This would be called by the script:

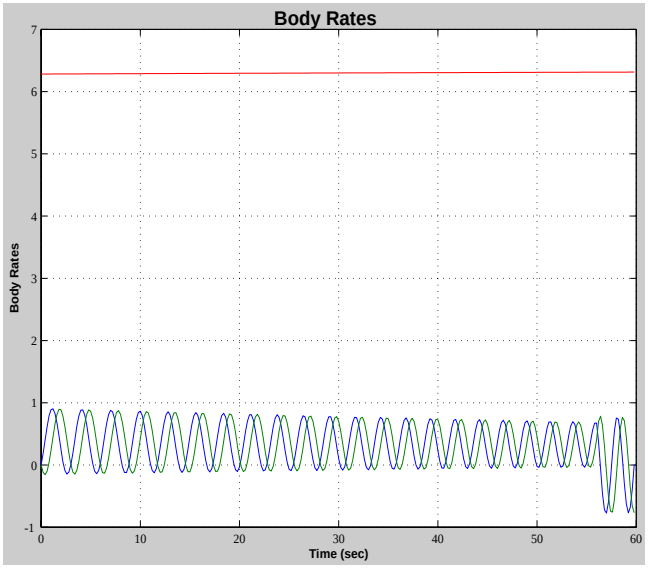
Initializing `xPlot` speeds up the script considerably. Notice the use of brack-

Listing 11-1 Rigid Body Attitude Simulation

```
dT          = 0.01;
nSimSteps   = 1000;
inertia     = [3000,0,0;0,1000,0;0,0,1000];
torque      = [0;0;0];
xPlot       = zeros(7,nSimSteps);
tPlot       = zeros(1,nSimSteps);
x           = [ [1;0;0;0]; [1;0;0] ];
t           = 0;
for k = 1:nSimSteps,
  xPlot(:,k) = x;
  tPlot(k)   = t;
  zIO(1:3,4) = [0;0;0];
  x          = RK4('ASim',x,dT,t, inertia,torque);
  t          = t + dT;
end
Plot2D(tPlot,xPlot,'Time (sec)',...
 ['Q1';'Q2';'Q3';'Q4';'Wx';'Wy';'Wz'],...
'Rigid Body')
```

ets in `x` to make the script easier to read. The results are shown in the following figure.

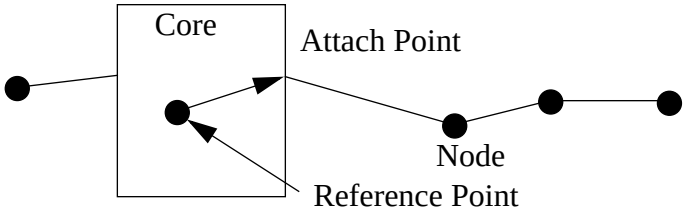
Figure 11-1 Rigid body simulation



Wire Model

The wire model simulates a spacecraft with deployable wires. Two models are included. One includes the extensional dynamics, modeling the stretching of the wire. The second applies kinematic constraints to control the length between wire nodes. The former can introduce very high frequency dynamics into the simulation but is useful if you expect the wire to stretch. The latter is useful for most scientific satellites with copper or aluminum wires. Either model can simulate deployment and both account for system center-of-mass motion. A spacecraft can have any number of wires. The configuration is illustrated below

Figure 11-2 Wire model



The following script illustrates the use of the model

Listing 11-2 Wire Simulation: Header

```
%-------------------------------------------------------------------------
% This script demonstrates the deployment of the wire from the
% spacecraft. This model assumes that the center-of-mass of the
% spacecraft does not move as the wires deploy. The simulation
% models the wire as a string of masses connected by springs.
% Orbit dynamics and gravity gradient are included.
%-------------------------------------------------------------------------
% Copyright © 1997 Princeton Satellite Systems, Inc. All rights reserved.
%-------------------------------------------------------------------------

% Clean up the workspace
%-----------------------
close all
clear all

% Constants
%----------
false  =  0;
true   =  1;
```

These parameters manage the script. kConst determines whether the constrained model is to be used.

Listing 11-3 Wire Simulation: Initialization

```
% Simulation parameters
%----------------------
tSim      = 120.0;
dT        = 0.125;
nSim      = tSim/dT;
nPlot     = min([tSim/dT 200]);
nPMax     = floor(nSim/nPlot);
nPlot     = floor(nSim/nPMax);
gGOn      = false;
kConst    = true;

% Print the time to go message every nTTGo steps
%-----------------------------------------------
nTTGo     = 1000;

% Time statistics function
%-------------------------
tToGoMem      = zeros(1,3);
ratioRealTime = 0;
```

The core body information. The model includes orbit dynamics.

Listing 11-4 Wire Simulation: Spacecraft properties

```
% Spacecraft properties
%----------------------
mass      = 800; % kg
r0        = [0;0;0];
inertia   = [104 0 0;0 107.8 0;0 0 125.4];

% Orbital elements [a i W w e M]
%-------------------------------
el            = [7000;0;0;0;0;0];
muEarth       = 3.98600436e5;

% Initial rigid body state
%-------------------------
[rECI, vECI]  = El2RV( el );
omega         = [0;0;0.5]*pi/30;
q             = [1;0;0;0];
torque        = [0;0;0]; % On the central body
force         = [0;0;0]; % On the central body
```

Create the wires. You can have as many wires as you like. Each column represents one wire. If you are using the constrained equations you can ignore the spring and damping constants.

Listing 11-5 Wire Simulation: Initialization of the wire model

```
% The wire model. Each column is one wire
%----------------------------------------
nNodes        = [ 3     3 ];
rWireBase     = [0 0;0.6 -0.6;0 0];
lWireMax      = [0.04 0.08];
massWire      = [0.4 0.4];
kSpring       = [ 3.0   3.0];  % Used only by WireFRB
cSpring       = [ 0.5   0.5 ]; % Used only by WireFRB
cDeploy       = cSpring;       % Used only by WireFRB
nodeDeploying = [ 0     0   ]; % To start undeployed set these to 3
vDeploy       = [ 0.001  0.001 ]; % m/sec
% Initialize the wire data structure
%-----------------------------------
[wireDS, x] = WireInit( nNodes, mass, massWire, lWireMax, kSpring,
cSpring, vDeploy, cDeploy, nodeDeploying, rWireBase, rECI, vECI, q, omega,
r0, inertia, gGOn );
```

Alpha is a constraint torque gain. mu determines the damping. omega the constraint stiffness. The number of iterations is generally less than 2.

Listing 11-6 Wire Simulation: Kinematic constraints

```
% If using the kinematic constraints
%-----------------------------------
penalty.alpha = 1e6;
penalty.mu    = 1;
penalty.omega = 10;
penalty.nIts  = 2;
```

Initialize the plotting arrays.

Listing 11-7 Wire Simulation: Initialization of plotting

```
% Plotting arrays
%----------------
xPlot = zeros(length(x),nPlot);
hPlot = zeros(1,nPlot);
tPlot = zeros(1,nPlot);
nP    = 0;
kP    = 0;

t     = 0;

% Initialize the status message function
%---------------------------------------
[ratioRealTime, tToGoMem] = TimeToGo( nSim, 0, tToGoMem, 0, dT );
```

The simulation loop. The first part shows the plotting functions.

Listing 11-8 Wire Simulation: Simulation loop plotting

```
for k = 1:nSim

  % Print the status message
  %-------------------------
  if( floor(k/nTTGo) == k/nTTGo )
    [ ratioRealTime, tToGoMem ] = TimeToGo( nSim, k, tToGoMem,
ratioRealTime, dT );
  end

  % Plotting
  %---------
  if( nP == 0 )
    kP           = kP + 1;
    xPlot(:,kP)  = x;
    hPlot(kP)    = WireH( x, wireDS );
    tPlot(kP)    = t;
    nP           = nPMax - 1;
  else
    nP           = nP - 1;
  end
```

This is the simulation. `WireDMch` simulates the deployment mechanism. It just assigns a nonzero extensional rate to the innermost wire segment.

Listing 11-9 Wire Simulation: Dynamics model

```
% Choose either the extensional stiffness or constrained wire models
%-------------------------------------------------------------------
  if( kConst == true )
    x = RK4( 'WireC',   x, dT, t, wireDS, muEarth, torque, force, penalty
);
  else
    x = RK4( 'WireFRB', x, dT, t, wireDS, muEarth, torque, force );
  end
  t           = t + dT;
  [wireDS, x] = WireDMch( wireDS, x, t );
end
```

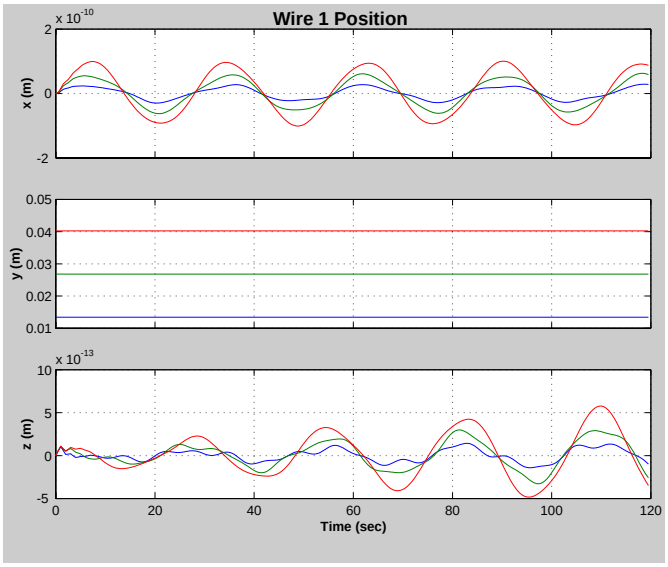
The following listing shows the plotting code.

Listing 11-10 Wire Simulation: Plotting

```
% Output
%-------
dOmega = [xPlot(11,:) - xPlot(11,1);...
          xPlot(12,:) - xPlot(12,1);...
          xPlot(13,:) - xPlot(13,1)];

magH = abs(hPlot(1));
hPlot = hPlot / magH;

Plot2D( tPlot, xPlot( 1: 3,:),   'Time (sec)', 'rECI (km)'        );
Plot2D( tPlot, xPlot( 4: 6,:),   'Time (sec)', 'vECI (km/sec)'    );
Plot2D( tPlot, xPlot( 7:10,:),   'Time (sec)', 'q'                );
Plot2D( tPlot, dOmega,           'Time (sec)', '∂Omega (rad/sec)' );
Plot2D( tPlot, hPlot - 1,        'Time (sec)', 'H/|H(0)| - 1'     );

WirePlot( xPlot, tPlot, wireDS );

disp(sprintf('Max momentum change = %12.4e with dT = %8.4f and tSim =
%8.4f',max(abs(hPlot-1)), dT, tSim))
```

The results of this wire simulation are shown in the following plot.

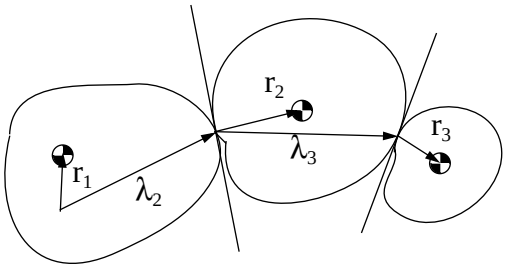
Figure 11-3 Wire simulation: Results



Tree Model

The tree model simulates multibody spacecraft arranged in a tree configuration.

Figure 11-4 Topological Tree



There can be no closed loops which also means that for any component exactly one hinges leads to the core. All of the subbodies are rigid and only rotational degrees of freedom are permitted at each hinge. In addition, the number of degrees of freedom at a hinge can be either 3 or 1. To simulate 2-dof hinge just interpose a massless and inertialess body.

The following script illustrates the use of the tree model.

Initialization of the model.

Listing 11-11 Tree Simulation: Initialization

```
%-----------------------------------------------------------------------
% © Copyright 1997 Princeton Satellite Systems, Inc. All rights reserved.
%-----------------------------------------------------------------------

% Vectors from previous body reference to the hinge of the body
%-------------------------------------------------------------
lambda1 = [0;0;0];
lambda2 = [2;0;0];
lambda3 = [1;0;0];

% Vector from body reference to body c.m. For all but the core
% the reference is always the hinge that leads to the core
%-------------------------------------------------------------
r1      = [0;0;0];
r2      = [0;0.5;0];
r3      = [0;0;0.5];

% Mass and inertia
%-----------------
m1      = 100;
m2      = 2;
m3      = 5;
i1      = diag([200 200 300]);
i2      = diag([  1   1   1]);
i3      = diag([  3   3   1]);
```

`TreeAdd` adds bodies to the link. You can give them a name.

Listing 11-12 Tree Simulation: Adding bodies.

```
% Add each body to the tree data structure
%------------------------------------------
body(1) = TreeAdd( i1, r1, lambda1, m1, 0, 0, [], [], [], [], [], 'Core'
);
body(2) = TreeAdd( i2, r2, lambda2, m2, 1, 2, [], [], [], [], [], 'Link'
);
body(3) = TreeAdd( i3, r3, lambda3, m3, 2, 3, [], [], [], [], [],
'Payload' );
```

Cause the inner link to acceleration.

Listing 11-13 Tree Simulation: A hinge acceleration.

```
% Internal torque
%----------------
body(2).torque = 0.1;
```

Define a low earth orbit.

Listing 11-14 Tree Simulation: A low earth orbit.

```
% Initial orbit
%--------------
r = [7000;0;0];
v = [0;sqrt(3.98600436e5/7000);0];
```

This initializes the tree data structures and the state vector. TreePrnt prints out all of the information in the data structure.

Initialize the multibody tree. and preallocate arrays.

Listing 11-15 Tree Simulation: Initialization

```
% Initialize the multibody tree
%------------------------------
[treeDS, x]  = TreeInit( body, r, v );

% Print out the tree
%-------------------
TreePrnt( body, treeDS );

% Plotting and initialization
%----------------------------
tSim        = 20;
dTSim       = 0.1;
nSim        = floor(tSim/dTSim);
hPlot       = zeros(1,nSim);
tPlot       = zeros(1,nSim);
xPlot       = zeros(length(x),nSim);
t           = 0;
```

Run the simulation. The angular momentum is computed for informational purposes only.

Listing 11-16 Tree Simulation: The simulation loop.

```
% Run the simulation
%-------------------
for k = 1:nSim

  % Save for plotting
  %------------------
  hPlot(k)   = Mag( TreeH( x, t, treeDS, body ) );
  xPlot(:,k) = x;
  tPlot(k)   = t;

  % Update the equations of motion
  %-------------------------------
  x = RK4( 'Tree', x, dTSim, t, treeDS, body );
  t = t + dTSim;
end
```

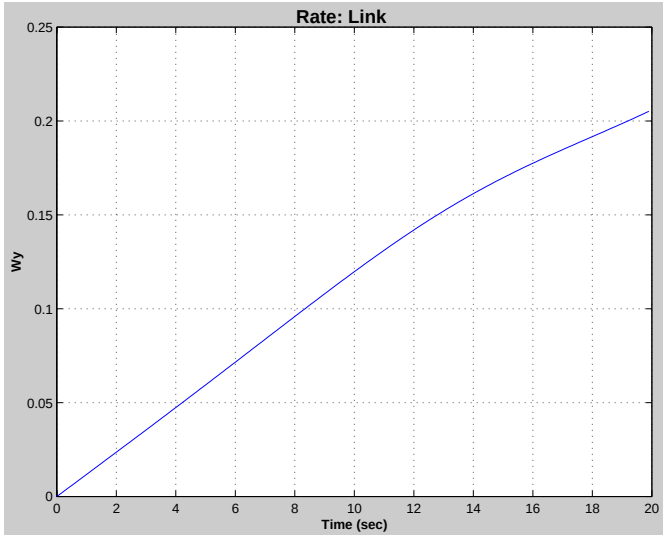
Plot the results.

Listing 11-17 Tree Simulation: Plotting.

```
TreePlot( tPlot, xPlot, treeDS, body, hPlot )
```

An example plot is shown below.

Figure 11-5 Tree Simulation: One of several plots.



Orbit Simulation

Introduction

The orbit models presented so far have assumed that the earth is a spherically symmetric body and that there are no other perturbations on the orbit. This is a good approximation for attitude control purposes, but not very good for orbit determination simulations. For more accurate orbit simulations use AEarth which generates the acceleration in spherical coordinates for a nonspherical earth gravitation model. JSph2Cart is the Jacobian that transforms spherical to Cartesian coordinates. The function RVOrbHF integrates the orbit equations using AEarth as the gravity model and can include the effects of the moon, sun, drag and other orbit perturbations.

NORAD Elements

NORAD orbital elements, also known as the NORAD two line elements, are used to represent the orbits of all spacecraft tracked by NORAD. The toolbox implements five different methods for propagating NORAD elements. An example of using the NORAD function is shown below. The five propagators

Listing 11-18 NORAD.m

```
%--------------------------------------------------------------------
%   Demo two of the five NORAD element propagators.
%--------------------------------------------------------------------
%   Copyright 1997 Princeton Satellite Systems, Inc.
%   All rights reserved.
%--------------------------------------------------------------------
rV = NORAD( 0, 1, 5, 'sgp', [], 'sgp.txt' );
t  = [0 360 720 1080 1440];

disp(sprintf('\nSGP\n'))
disp('Position')
for k = 1:5
  disp(sprintf('%11.2f %11.2f %11.2f %11.2f',t(k), rV.r(:,k)));
end
disp(sprintf('\nVelocity'))
for k = 1:5
  disp(sprintf('%11.2f %11.2f %11.2f %11.2f',t(k), rV.v(:,k)));
end
```

Listing 11-18 NORAD.m

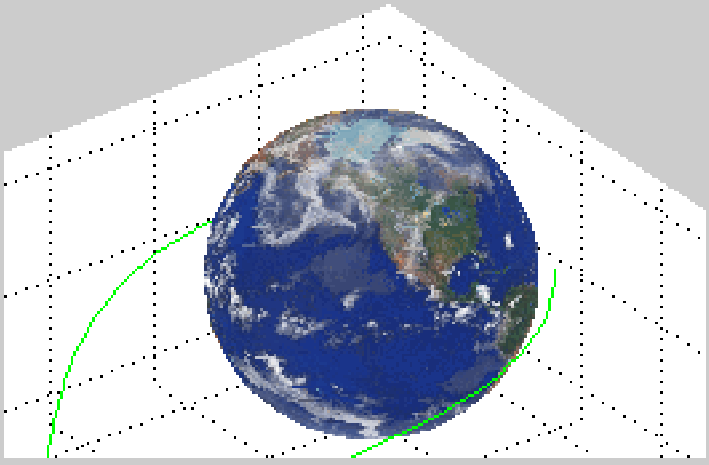
```
disp(sprintf('\nSDP8\n'))
rV = NORAD( 0, 1, 5, 'sdp8', [], 'sdp.txt' );

t  = [0 360 720 1080 1440];
disp('Position')
for k = 1:5
  disp(sprintf('%11.2f %11.2f %11.2f %11.2f',t(k), rV.r(:,k)));
end
disp(sprintf('\nVelocity'))
for k = 1:5
  disp(sprintf('%11.2f %11.2f %11.2f %11.2f',t(k), rV.v(:,k)));
end
```

```
SGP

Position
       0.00     2328.97    -5995.22     1719.98
     360.00     2456.01    -6071.94     1222.96
     720.00     2567.40    -6112.50      713.97
    1080.00     2663.03    -6115.37      195.74
    1440.00     2742.86    -6079.14     -328.86

Velocity
       0.00        2.91       -0.98       -7.09
     360.00        2.68       -0.45       -7.23
     720.00        2.44        0.10       -7.32
    1080.00        2.20        0.65       -7.36
    1440.00        1.95        1.21       -7.35


SDP8

Position
       0.00     7466.26      419.45     5833.52
     360.00    -3324.53    32340.61   -24675.30
     720.00    14228.24    24233.73    -4858.57
    1080.00   -10138.27    22216.15   -23409.87
    1440.00     9430.25    33839.76   -15399.16

Velocity
       0.00        5.11        6.44       -0.18
     360.00       -1.30       -1.16       -0.28
     720.00       -0.34        2.65       -2.08
    1080.00       -1.00       -2.33        0.77
    1440.00       -1.12        0.85       -1.50
```

are SGP, SGP4, SGP8, SDP4, SDP8. The first three are suitable for low earth orbiting satellites, the last two for high altitude satellites. All functions read in from files in the NORAD format which are available on-line.

CHAPTER 6

Orbit Propagation



This chapter shows you how to propagate orbits and perform related tasks.

Orbit Propagation

Introduction

The function `PropagateOrbitPlugin`, allows for 3 degree-of-freedom propagation of orbits. It can be used interactively or in batch mode. It has the following features:

- User selectable gravity models. GEM-T1, JGM-2, JGM-3 and WGS-84 are supported. Other models can be downloaded.
- Ballistic propagation of covariance matrix for state vector. A plug in function allows the user to control the plant noise covariance matrix.
- Propagate in ECI, ECR (earth-fixed) or ECIR. The last is an inertial frame that is coincident with EF at time 0.
- The user has complete control accuracy of numerical integration. The propagator uses the MathWorks state-of-the-art ode113 propagator.
- The user can select the time step. The time step can be variable and negative. It can be defined by any Matlab expression or function.
- The user can specify any one of over thirty stopping conditions.
- Flight path angle, altitude, geodetic latitude and longitude are automatically computed.
- Outputs can be saved to a mat file for further analysis.
- Sun/moon/earth perturbations can be added if you are propagating in an earth-centered or moon-centered frame

Limitations

The following are some limitations that will be addressed in the next version:

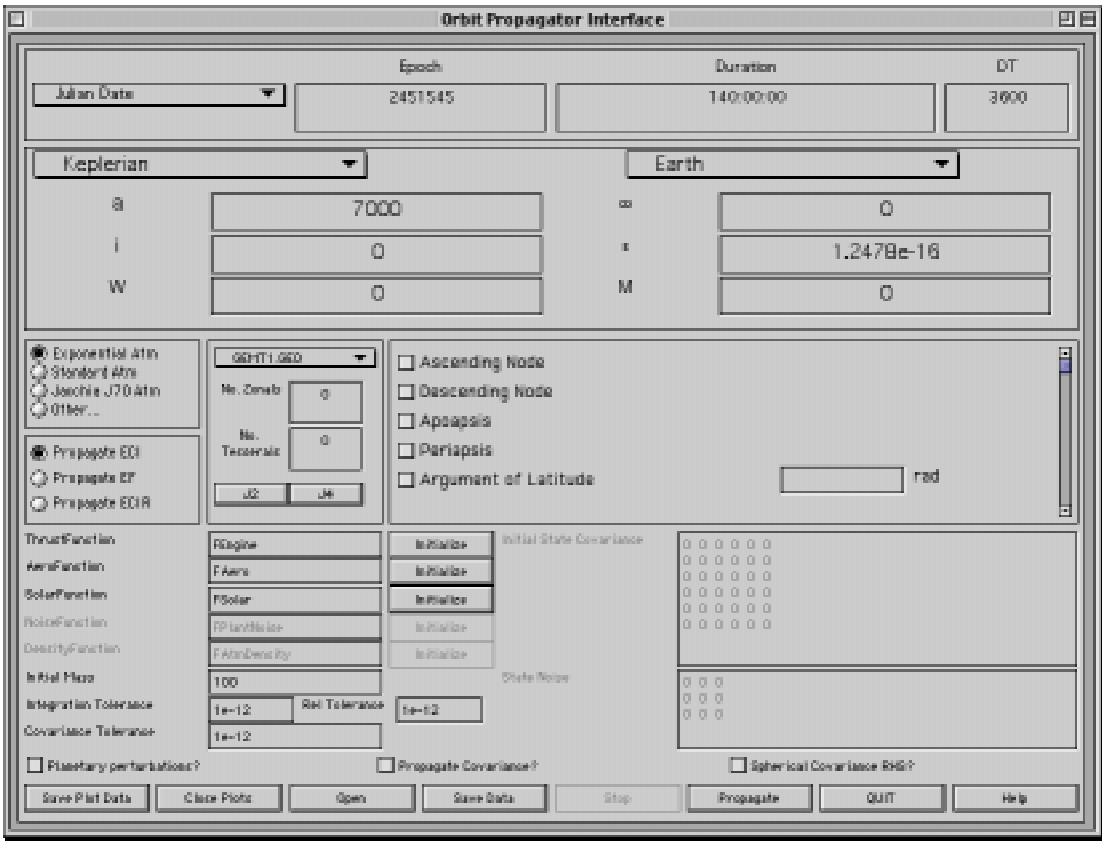
- The propagator does not switch centers when the trajectory leaves one planet’s sphere of influence and enters another.

- Only the moon and sun are added as perturbations when the earth is the center and the sun and earth are entered as perturbations when the moon is the center.
- Only centers for which the toolbox has pictures are included. This leaves out many moons.
- Only Mean-of-Aries 2000, Earth-Fixed and inertial coincident with earth-fixed frames are available.
- The noise function is not computed automatically.

Using the Graphical User Interface

The graphical user interface displays all options for using the propagator.

Figure 6-1 Orbit Propagation GUI



Overview

The orbit propagator interface, which is created by PropagateOrbitPlugin.m allows you to propagate any type of orbit using ode113, the MathWork's state-of-the-art propagator.

The GUI has several different “panes.” These are summarized below. The following sections discuss each in more detail.

- Time Pane—set the time
- Elements Pane—set the initial orbital elements. These are always referenced to the ECI frame
- Atmosphere Pane — select one of four atmosphere models
- Propagation Pane —select the coordinate frame in which to propagate
- Gravity Model Pane select the gravity model and the order of the model
- Stopping Conditions Pane select the stopping condition for the simulation
- Customization Pane — select simulation options and functions you wish to plug into the simulation
- Buttons —control the simulation

The propagator can be run interactively or as part of a Matlab script

Panes

Time Pane

The time plug-in, in the upper right corner of the Propagator window, contains the values for the epoch, maximum duration, and time step of the next propagation run. This plug-in allows you to convert the epoch between Julian Date and UTC Calendar date. The duration field allows you to enter a maximum duration for the run. If the specified stopping condition is not met within the specified duration the run will stop. The default duration is one hour. The DT field allows you to specify the time step of the propagation. The default DT is 100 seconds. DT may be negative.

A set of times can be entered for the duration. For example you could enter:

```
0:25 hr
```

or

```
linspace(0,60000,1000) sec
```

or

```
[ 0 1 7 9 11 22 43 900] min
```

or

```
[ 900 43 22 11 9 7 1 0] hr
```

or

```
MyTimeSequence hr
```

or

```
3 orbits
```

Any Matlab command or function can be entered into duration. If anything except a time is entered, DT is ignored. The output will be computed at these specific times only. The numerical integration routine will select the appropriate time step to achieved the specified numerical accuracy independent of the entered times.

Elements Pane

The elements plug-in allows you to enter and convert between Keplerian elements, equinoctial elements, and RV elements. The planet used for the orbit center is also shown. Element Sets are given in the following table.

Table 6-1 Element sets

Keplerian		RP/RA		Equinoctial		RV
a	Semi-major axis	R_p	Perigee radius	a	Semi-major axis	r_1
i	Inclination	i	Inclination	P_1		r_2

Table 6-1 Element sets

Keplerian		RP/RA		Equinoctial		RV
Ω	Right Ascension of the Ascending N ode	Ω	Right Ascension of the Ascending N ode	P_2		r_3
ω	Argument or Perigee	ω	Argument or Perigee	Q_1		v_1
e	Eccentricity	R_a	Apogee radius	Q_2		v_2
M	Mean anomaly	M	Mean anomaly	l	Mean longitude	v_3

The RV elements are position and velocity vector.

Atmosphere Pane

You can select any one of four models. The first two are altitude dependent and are based on a spherical planet model. The third will enable the default Atmospheric density plugin function, `FAtmDensity`, which implements the Jacchia J70 model. Other... allows you to enter your own atmosphere model.

Propagate Pane

This selects the frame in which to propagate. The ECIR frame is an inertial frame which coincides with the EF frame at the start of the simulation.

Gravity Model Pane

The Database model is a spherical planet model with the gravitational parameter extracted from the Constant function. Currently, this the only model you can use for planets or moons other than the earth.

You can download GEMT*, JGM-* and WGS84 models from various websites. Any file with the suffix.GEO will be displayed in the pull-down menu. You can enter the number of zonal and tesseral harmonics you wish to use. If you just want the J2 or J4 zonal, you can hit the button to set the appropriate model order.

Stopping Conditions Pane

You can select any combination of stopping conditions listed in the pane. For each you can enter a tolerance. Some require data. The stopping conditions are

described in the following table. Some of the conditions aren't relevant to true

Table 6-2 Stopping Conditions

Stopping Condition	Explanation
Impact	Stop when orbit crosses the central body ellipsoid. This is the same as setting the geodetic altitude to zero.
Ascending Node	Stop at the ascending node.
Descending Node	Stop at the descending node.
Apoapsis	Stop at apoapsis.
Periapsis	Stop at periapsis
Argument of Latitude	Stop when $\omega + \nu$ = value
XZ Crossing	Stop when ECI y = 0
XY Crossing	Stop when ECI z = 0
YZ Crossing	Stop when ECI x = 0
True Anomaly	Stop when ν = value
Approaching Distance	Stop when r = value and is decreasing
Receding Distance	Stop when r = value and is increasing.
Increasing Velocity	Stop when v = value and is decreasing
Decreasing Velocity	Stop when v = value and is increasing.
Flight Path Angle	Stop when the flight path angle = the value
Perigee Height	Stop when perigee radius = value†
Apogee Height	Stop when apogee radius = value†
Geodetic Latitude	Stop when geodetic latitude = value
Geocentric latitude	Stop when geocentric latitude = value
Geodetic Altitude	Stop when geodetic altitude = value
Semi major Axis	Stop when a = value†
† Only relevant to non Keplerian orbits.	

Table 6-2 Stopping Conditions

Stopping Condition	Explanation
Mean Anomaly	Stop when M = value
Orbital Period	Stop when orbital period = value†
Right Ascension of Ascending Node	Stop when Ω = value and only at the descending node†
Right Ascension of Descending Node	Stop when Ω = value and only at the ascending node†
Change in Inertial Velocity Magnitude	Stop when |v| = value
Vehicle Longitude	Stop with earth-fixed longitude = value
Argument of Perigee	Stop when argument of perigee = value†
Eccentric Anomaly	Stop when E = value
† Only relevant to non Keplerian orbits.	

Keplerian orbits in which the orbital elements do not change.

Customization Pane

The first four buttons are the names of functions the propagator should call. The default names in the GUI are default functions. When you hit initialize, the function will display an initialization GUI.

The Noise Function returns the plant noise matrix, q, for the covariance propagation:

pDot = f*p + p*f' + q

where f is the matrix of partials of the right-hand-side of the orbit.

The initial mass is the mass of the spacecraft.

The next two parameters relate to the integration accuracy

Rel Tolerance gives the relative error tolerance The estimated error in each integration step satisfies

```
e(i) <= max(RelTol*abs(y(i)),Tol(i)).
```

Integration Tolerance gives the absolute error tolerance which applies to all components of the solution vector.

The next two checkboxes allow you to select planetary perturbations and/or to propagate the covariance.

When you select planetary perturbations you will get the moon and the sun if the earth is selected as the center through the elements pane or the earth and the sun if the moon is selected as the center. If you select other planets or moons you will not get any perturbations.

When you select Propagate Covariance the initial covariance input will be enabled. You should enter the initial state covariance into the box. You need only enter upper or lower triangular information. The propagator automatically does

```
p = 0.5*(p + p')
```

to insure that the covariance matrix is symmetric. You have the option to select Spherical Cov. This uses a spherical earth model just to propagate the covariance. This is much faster than using the full model.

Buttons

- Save Plot Data saves the plot data in a mat file of type *.OPP. The plot is saved as a matrix with rows:

  ```
  [ x;y;z;vX;vY;vZ;mass;pXX;....pVZVZ;pXY;pYZ...]
  ```

  p is saved in columns starting with the main diagonal and moving up. For example, a 3x3 matrix would be saved as

  [a11;a22;a33;a12;a23;a13];

- Close Plots—Close the plot windows.
- Open—Open a gui dat file of type *.OPG.
- Save Data—Save the gui data in a mat file of type *.OPG.
- Propagate—Run the propagator.

- QUIT—Quit. It will ask you to save unsaved plot and GUI data.
- Help—Get help.

Setting Up a Batch Script

You can run `PropagateOrbitPlugIn` from a script. This allows you to set up large numbers of cases and run them automatically. OPDemo shows how to set up a batch run

```
d = load('OPDemo.mat');
d.epoch.dT = 3600;
tag = PropagateOrbitPlugin( 'initialize' );
      PropagateOrbitPlugin( 'set data', tag, d );
      PropagateOrbitPlugin( 'propagate', tag );
  pause
      PropagateOrbitPlugin( 'close plots', tag );
      PropagateOrbitPlugin( 'quit', tag );
```

In the first line a saved data file is loaded. This file was created and saved using the GUI interactively, In the second the time step is customized. Any field in the data structure may be customized. In the third step the GUI is initialized. In the fourth the datastructure d is loaded. The orbit is then propagated. Plots are closed after the pause and the final step is to quit. `tag` identifies the GUI.Possible commands are listed in the following table.

Table 6-3 PropagateOrbitPlugin API

Prototype	Purpose
`tag = PropagateOrbitPlugin( 'initialize' );`	Initialize the plugin.
`PropagateOrbitPlugin( 'set data', tag, d );`	Set the data in the GUI
`PropagateOrbitPlugin( 'propagate', tag );`	Propagate the orbit.
`PropagateOrbitPlugin( 'close plots', tag );`	Close the plots
`PropagateOrbitPlugin( 'save', tag, fileName );`	Save the GUI data.

Table 6-3 PropagateOrbitPlugin API

Prototype	Purpose
`PropagateOrbitPlugin( 'save plot data', tag, fileName );`	Save the plot data
`PropagateOrbitPlugin( 'quit', tag );`	Close the GUI.

Plugin Functions

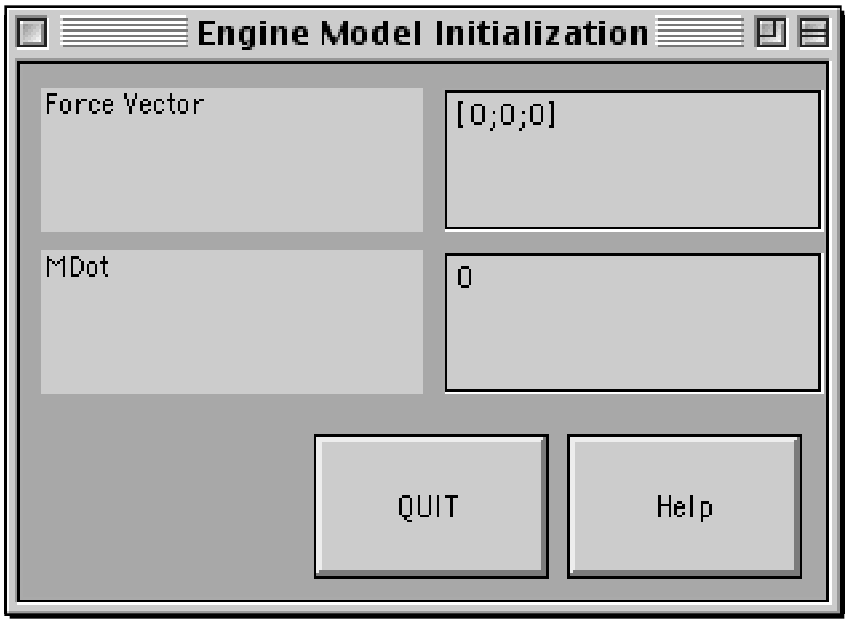
The orbit propagator has four plugin in functions that allow you to customize the simulation. When you first bring up the GUI four default functions are shown. They are described in the following table.

Table 6-4 Plugin Functions

Function	Description
`FEngine`	Computes engine force and spacecraft mass. The force is in the LVLH frame.
`FAero`	Computes aero forces in the ECI frame.
`FSolar`	Computes solar forces in the ECI frame.
`FPlantNoise`	The Q matrix in the covariance propagation equation.
`FAtmDensity`	Atmospheric density. The baseline model is Jacchia J70.

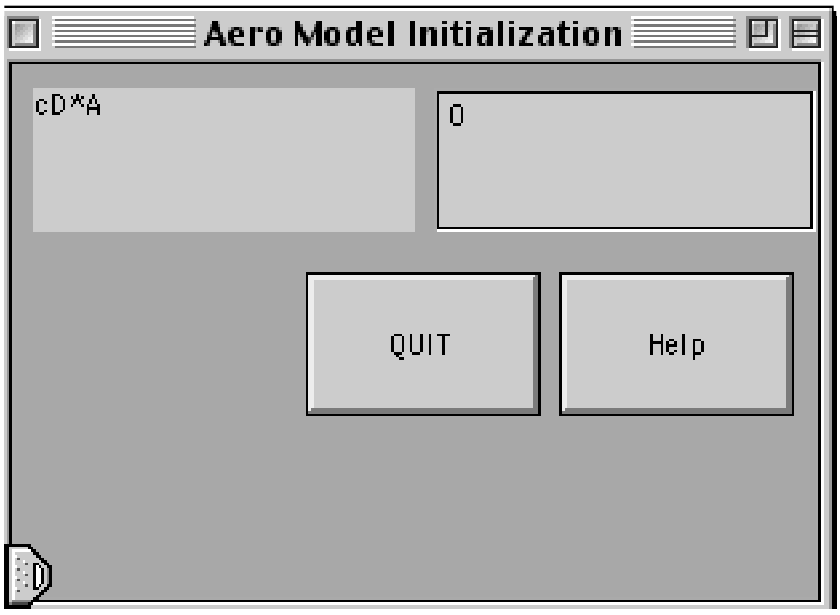
Each model supplied with the toolbox has its own GUI. These popup when you hit the Initialize button next to each function's name.

Figure 6-2 `FEngine` GUI



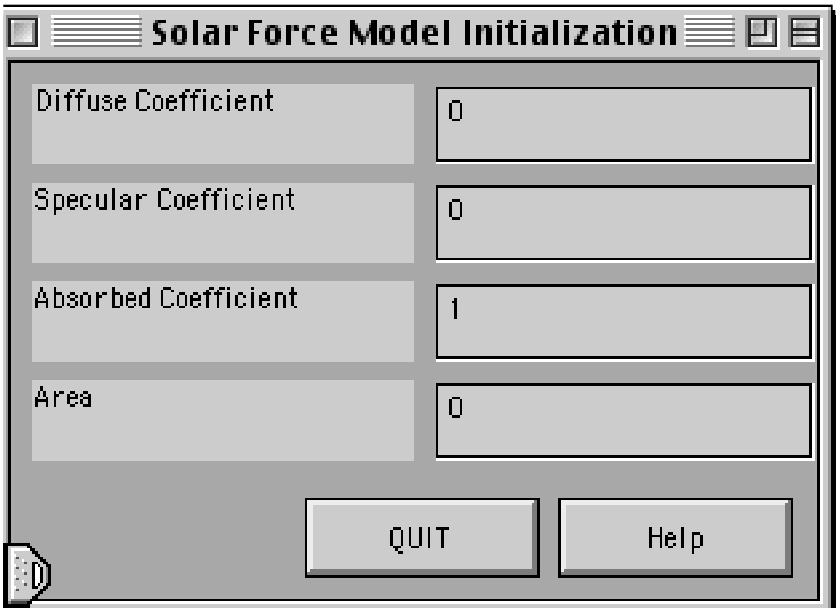
The `FEngine` model allows you to enter a constant force in the ECI frame and a constant mass change rate.

Figure 6-3 `FAero` GUI



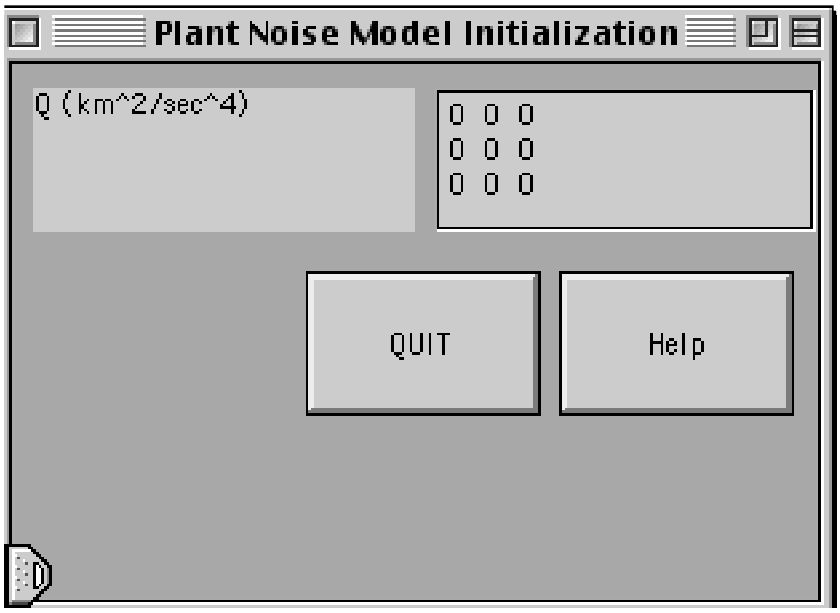
The FAero model assumes a flat plate spacecraft with the plate always normal to the velocity vector. The input is the product of the drag coefficient and plate area.

Figure 6-4 FSolar GUI



The FSolar model is a flat plat model in which the plate is always normal to the sun vector.

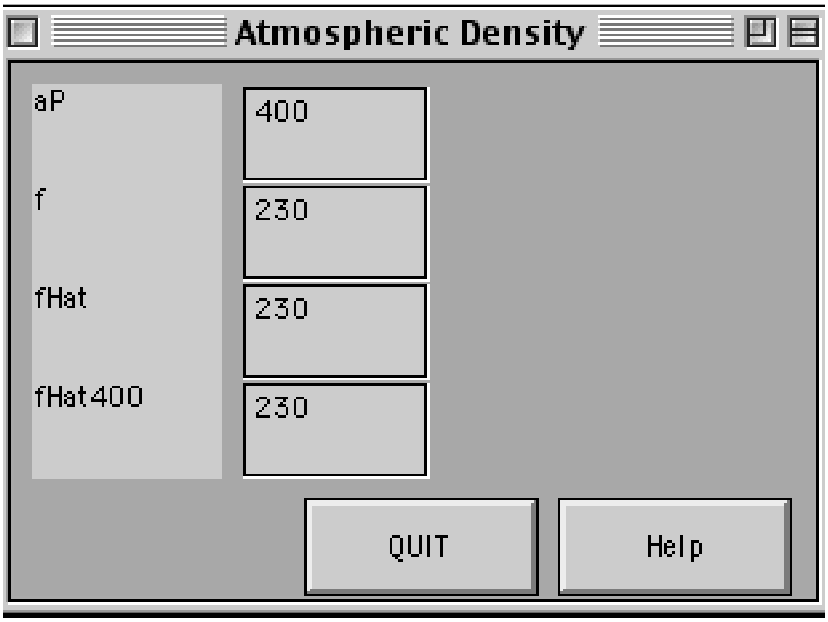
Figure 6-5 `FPlantNoise` GUI



`FPlantNoise` function assumes a constant plant noise covariance matrix. The inputs are the uncertainty in body accelerations.

The `FAtmDensity` GUI gives inputs for the Jacchia J70 model.

Listing 6-1 `FAtmDensity` GUI



The J70 inputs are given in the following table.

Table 6-5 Jacchia J70 Atm inputs.

Input	Description
aP	Geomagnetic index 6.7 hours before the computation
f	Daily 10.7 cm solar flux (e-22 watts/m^2/cycle/sec)
fHat	81-day mean of f (e-22 watts/m^2/cycle/sec)
fHat400	fHat 400 days before computation date

Building a Plugin Function

All plugin in functions have the same structure. In this section, we will walk you through the structure of `FAero`, the simplest function.

Listing 6-2 on page 49 shows the function call list and the switch statement that makes up the main function. action is the argument for the switch statement and modifier is the GUI tag, which identifies the GUI.

Listing 6-2 FAero: Main Function

```
function x = FAero( action, modifier, u, t )

if( nargin < 1 )
  action = 'initialize';
end

switch action

  case 'initialize'
    x = Initialize( modifier );

  case 'update'
    x = Update( modifier, u, t );

  case 'get data'
    x = GetData( modifier );

  case 'store data'
    h   = GetH( modifier );
    h.d = GetData( modifier );
    PutH( h );

  case 'set data'
    SetData( modifier );

  case 'help'
    HelpSystem( 'initialize', 'SCHelp' );

  case 'quit'
    h = GetH( modifier );
    PropagateOrbitPlugin('close function',h.propagateOrbitPluginTag,'FAero');
    CloseFigure( h.fig )

end
```

The `Initialize` subfunction draws the GUI and returns the tag for the fig-

Listing 6-3 FAero: `Initialize`

```
function tag = Initialize( propagateOrbitPluginTag )

name      = 'Aero Model Initialization';
position  = [ 5 5 300 200];
tag       = GetNewTag( name );
h.fig     = figure( 'position', position,'color',[ 0.66 0.66
0.66], 'NumberTitle', 'off', 'name', name, 'resize', 'off',
'tag', tag );
fontSize  = position(4)*9/185;
v        = {  'parent', h.fig, 'fontunits', 'pixels', 'fontsize',
fontSize, 'horizontalalignment','left'};
space     = position(3)/40;
x0        = space;
dX        = position(3)/2;
xW        = dX - space;
dY        = (position(4) - 2*space)/3;
yW        = dY - space;

x         = x0;
y         = position(4) - dY;

% The orbit propagator tag
%-------------------------
h.propagateOrbitPluginTag = propagateOrbitPluginTag;

% Inputs
%-------
h.gui.cDAText    = uicontrol( v{:}, 'position', [ x y xW yW],
'string', 'cD*A', 'style', 'text' ); x = x + xW + space;
h.gui.cDA        = uicontrol( v{:}, 'position', [ x y xW yW],
'string', '0',    'style', 'edit' ); x = x0; y = y - dY;

% GUI controls
%-------------
xW               = (position(3) - 5*space)/3;
x                = position(3) -  xW - space;
y                = y - space;

h.gui.help       = uicontrol( v{:}, 'position', [ x y xW yW],
'string', 'Help', 'callback', CreateCallback( 'help', tag ) );  x
= x - xW - space;
h.gui.quit       = uicontrol( v{:}, 'position', [ x y xW yW],
'string', 'QUIT', 'callback', CreateCallback( 'quit', tag ) );  x
= x - xW - space;

PutH( h )
```

ure. The Inputs and GUI controls lines show you how to write uicontrols for

edit boxes seen in the GUI. `uicontrols` require you to pass them parameters in pairs. To simplify the code we put a lot of the parameters in the cell array `v`. The `CreateCallback` subfunction generates a call back to the `FAero` function. For example the last line creates the call:

```
FAero( 'quit', tag )
```

Listing 6-4 `FAero: CreateCallback`

```
function s = CreateCallback( action, modifier )

s = [ 'FAero( '''action ''','''' modifier ''')'];
```

See the Matlab documentation on how to build GUIs for more information.

gets data stored in the GUI and set data sets the data. `SetData` is used by

Listing 6-5 `FAero`: Getting and setting data.

```
function d = GetData( modifier )

h = GetH( modifier );

d.cDA = str2num( get( h.gui.cDA, 'string' ) );

function SetData( modifier, d )

h = GetH( modifier );

set( h.gui.cDA,  'string', num2str( d.cDA ) );
```

the orbit propagation function when you pass it the saved propagator data structure. `GetData` is used internally as shown in the next listing.

The `Update` subfunction computes the aerodynamic drag. It gets the figure handle, gets the data from the figure handle, and combines it with the velocity stored in data structure `u`, to compute the force.

Listing 6-6 `FAero`: The `Update` subfunction.

```
function x = Update( modifier, u, t )

h   = GetH( modifier );
d   = GetData( modifier );

x   = -0.5*d.cDA*u.rho*Mag(u.v)*u.v*1e3;
```

Reference Frames

Coordinate Transformations

`CoordinateTransform` transforms between:

- ECI
- ECR
- Geodetic latitude, longitude and altitude

For example

```
r = [ 6524.834;6862.875;6448.296];

x = CoordinateTransform( 'eci', 'ef', r, 2449773 )

x =

   1.0e+03 *
   2.85101441662010
   9.03243563995045
   6.44514208556912

x = CoordinateTransform( 'eci', 'llr', r, 2449773 )

x =

   1.0e+03 *
   0.00059923067422
   0.00126505172630
   5.08520910899186
```

Finding ECI/ECR coincidence

`EarthRotationZero` finds a Julian date for which the Greenwich Apparent Sidereal Time (GAST) is zero. For example

```
x = EarthRotationZero( JD2000 )

x =

   2.5047e+06
```

gives a Julian Date within a day of JD2000. For this date

```
GASTime( x )

ans =

   7.2978e-09
```

You can also find the transformation matrix from ECI to ECR at any time by typing

```
m = ECIToECIR( jD )
```

`m` transforms from ECI to the frame aligned with EF at time `jD`. This constant offset frame can be used to propagate equations in the ECIR frame.

Sun

`SunV1` provides a moderate precision (0.01 deg) sun angle.

```
uSun = SunV1( jD );

[ rA, dec] = U2RADec( uSun )
```

The second function gives the right ascension and declination for the sun.

Interfacing to STK

You can interface to STK using the function `STKOrbit`. `STKOrbit` allows you to exchange data with STK by creating an input file for STK.

```
[ err, message] = STKOrbit( fileName, ver, epoch,
nPoints, time, position, velocity, type )
```

Table 6-6 I/O for STKOrbit

Variable	Description
fileName	Filename for output. Will be overwritten if it already exists.
ver	STK version number (string!)

Table 6-6 I/O for STKOrbit

Variable	Description
epoch	Beginning date and time: [Y M D H M S]
nPoints	Number of orbit data points
time	Time in seconds for each data point
position	Position data in km (if type = ECF or ECI) or in [degrees,degrees,km] (if type = LLR)
velocity	Velocity data in km/sec (if type = ECF or ECI) or [degrees,degrees,km]/sec (if type = LLR)
type	'ECI','ECF',or 'LLR'
err	err ~= 0 means an error occurred
message	Error message

Ground Coverage

Ground coverage is found using

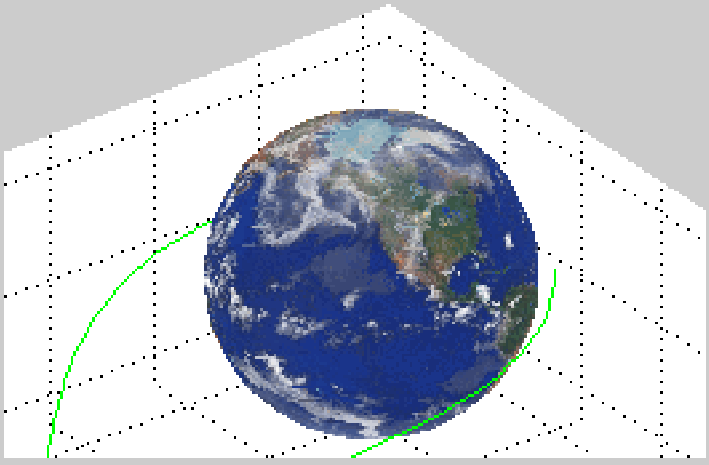
```
x = GroundCoverage( xECI, jD, d )
```

where `xECI` is the ECI state vector `[ r; v]`, `jD` is the Julian date and `d` is the data structure in which `d.fOV.x` is the x axis field-of-view, `d.fOV.y` is the y-axis field-of-view and `d.m` transforms from the body frame to the sensor frame. The sensor is assumed to have its boresight along +z in the sensor frame. If `d.m` were the identity matrix the sensor would point along +z in the body frame. For this function, the satellite body frame is assumed to be aligned with the LVLH frame.

`d.e` is a 3-by-1 vector with the [x dimension;y dimension;z dimension] of the ellipsoid. If the planet is spherical the three dimensions are the same.

CHAPTER 13

Budgets



This chapter discusses how to generate pointing and fuel budgets. These budgets are some of the most important elements of spacecraft system design.

Pointing Budget

PBudget generates antenna beam pointing budgets. Its inputs are a n-by-3 matrix of error contributions, an n-by-m matrix of categories, an n-by-m matrix of descriptions, and a 2-by-1 matrix of antenna offsets. For the category and description matrices m signifies the length of the longest string. PBudget

Listing 13-1 PBudget Example

```
errors = [0.01 0.02 0.03;...
          0.02 0.04 0.01;...
          0.04 0.03 0.05;...
          0.05 0.00 0.06]; % degrees
categ  = 'Bias';
categ  = str2mat(categ,'Bias');
categ  = str2mat(categ,'Diurnal');
categ  = str2mat(categ,'Diurnal');
desc   = 'Thermal';
desc   = str2mat(desc,'Misalignments');
desc   = str2mat(desc,'Thermal');
desc   = str2mat(desc,'Misalignments');
aZ     = 0.0;
eL     = 0.0;
[cep,r,s,t] =
PBudget(errors,categ,desc,aZ,eL,'MyBudget');
```

always adds categories and combines errors within categories by taking the square root of the sum of the squares. You can have as many categories as you wish. The descriptions have no effect on the computations.

The outputs are the 3-sigma circular error, the azimuth and elevation beam pointing errors, the category totals and the totals. The 3-sigma error is the angular error that there is a 0.98655 probability that the beam center is within. If the last input is given, PBudget will create a pointing budget file called MyBudget

The circular error is computed by numerically integrating the two-dimensional probability density function for azimuth and elevation. The Matlab quad routine is used to perform the inner integration. A bisection search is used to find the value of angular radius with that marks the 3-sigma boundary. A complete pointing budget is given in the script TPBudget.

Fuel Budgets

Computation of fuel budgets is a straightforward bookkeeping exercise. The most important point to remember is that the fuel consumption of the thrusters is a function of system pressure. Consequently, fuel budgets must be done chronologically.

FBudget generates fuel budgets. There are four types of maneuvers supported. They are spin precession maneuvers, spin changes, AKM firings, and station-keeping maneuvers. You set up each maneuver using the FBudget data structures as shown in the following example.

Listing 13-2 Fuel Budget: Moment Arms

```
% You need position vectors and moment arms.
r = [ -0.8 -0.8 -0.8 -0.8  0.8  0.8  0.8  0.8  0.8 -0.8 -0.8  0.8  0.2  -
0.2  -0.2  0.2;...
      -0.9 -0.8  0.7  0.8 -0.8 -0.8  0.8  0.8 -0.8 -0.9 -0.8 -0.7 -0.8  -
0.8  -0.8 -0.8;
       0.8 -0.8 -0.8  0.8  0.9 -0.8 -0.8  0.7  0.8  0.8 -0.8 -0.8  0.15 -
0.2 -0.2  0.2];

u = [ -1  -1  -1  -1   1   1   1   1   0   0   0   0   0   0   0   0;...
       0   0   0   0   0   0   0   0   1   1   1   1   1   1   1   1;...
       0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
```

In the next listing the thruster performance data is collected into data structures.

Listing 13-3 Fuel Budget: Thruster performance data

```
% Collect thruster data into the data structures.
for k = 1:12
f(k).iSP      = [100 100 10]; % See REA.m for definition of this array
f(k).thrust   = 2.07e-6; % See REA.m for definition of this array
f(k).riseTime = 0.016; % In seconds
f(k).fallTime = 0.016;
f(k).u        = u(:,k);
f(k).r        = r(:,k);
f(k).systemID = Odd(k) + 1; % You can have as many fuel systems as you
want
end

for k = 13:16
f(k).iSP      = [100 250 10];
f(k).thrust   = 0.828e-6; % In this case just multiplying this by pressure
f(k).riseTime = 0.016;
f(k).fallTime = 0.016;
f(k).u        = u(:,k);
f(k).r        = r(:,k);
f(k).systemID = Odd(k) + 1;
end
```

The next listing has general spacecraft data

Listing 13-4 Fuel Budget: General spacecraft data

```
% Assemble general spacecraft data.
x.regulatedPressure = [250 250]*6895;
x.systemTemp        = [298 298];
x.systemVol         = [1.6505e-01 1.6505e-01];
x.systemFuel        = [1.1789e+02 1.1789e+02];
x.pressurantMass    = [1.8372e-01 1.8372e-01];
x.pressurantR       = MolWt2R( 0.004 );
x.dryMass           = 9.5514e+02;
x.fuelDensity       = 1000;
x.thrusterData      = f;
x.controlDT         = 1;
```

The following listing shows the initialization of the arrays.

Listing 13-5 Fuel Budget: Array initialization

```
% Event 1: Initialization
%------------------------
k = k + 1;
event(k).type         = 'initial';
event(k).description  = 'Initial State';
event(k).aCSThruster   = [];
event(k).dVThruster    = [];
event(k).dVTotal       = [];
event(k).dHTotal       = [];
event(k).disturbance   = [];
event(k).pulsewidthACS =  0;
event(k).pulsewidthDV  =  0;
event(k).nPulses       = [];
event(k).duration      = [];
event(k).regulated     =  0;
event(k).cM            = [0;0;0];
```

The next listing shows the entry for a continuous Delta-V burn.

Listing 13-6 Fuel Budget: Continuous delta-v burn

```
% Event 2: A continuous delta-V burn using different ACS and DV thrusters
%-------------------------------------------------------------------------
k = k + 1;
event(k).type          = 'delta-v';
event(k).description   = 'Delta V Continuous';
event(k).aCSThruster   = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster    = [13 14];
event(k).dVTotal       = 100; % Total delta V in m/sec
event(k).dHTotal       = [];
event(k).dTControl     = 1;
event(k).disturbance   = [];
event(k).nPulses       = [];
event(k).duration      = [];
event(k).regulated     =  0; % If not regulated use blowdown curves
event(k).cM            = [0;0;0];
```

The next listing shows the entry for a delta-v burn with off-pulsing.

Listing 13-7 Fuel Budget: Delta-v with off-pulsing

```
% Event 3: Delta-V off-pulsing some thrusters
%-----------------------------------------
k = k + 1;
event(k).type          = 'delta-v';
event(k).description   = 'Delta V Off Pulse';
event(k).aCSThruster   = [1 2 3 4 9 10 11 12];
event(k).dVThruster    = [1 2 3 4];
event(k).dVTotal       = 100;
event(k).dHTotal       = [];
event(k).dTControl     = 1;
event(k).nPulses       = [];
event(k).duration      = [];
event(k).regulated     =  0;
event(k).cM            = [0;0;0];
```

The next listing shows an unloading momentum entry.

Listing 13-8 Fuel Budget: Unloading momentum

```
% Event 4: Unloading momentum
%---------------------------
k = k + 1;
event(k).type         = 'delta-h';
event(k).description  = 'Delta H';
event(k).aCSThruster   = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster    = [];
event(k).dVTotal       = [];
event(k).dHTotal       = [1;0;0];
event(k).dTControl     = 1;
event(k).disturbance   = [];
event(k).nPulses       = [];
event(k).duration      = [];
event(k).regulated     =  0;
event(k).cM            = [0;0;0];
```

Listing 13-9 Fuel Budget: Using ACS thrusters

```
% Event 5: Using thrusters for ACS
%--------------------------------
k = k + 1;
event(k).type         = 'acs';
event(k).description  = 'ACS Disturbance';
event(k).aCSThruster   = [1 2 3 4 5 6 7 8 9 10 11 12];
event(k).dVThruster    = [];
event(k).dVTotal       = [];
event(k).dHTotal       = [];
event(k).dTControl     = 1;
event(k).disturbance   = [0;1;0];
event(k).nPulses       = [];
event(k).duration      = 10000;
event(k).regulated     =  0;
event(k).cM            = [0;0;0];
```

The next listing shows an example of firing a few pulses

Listing 13-10 Fuel Budget: Firing a few pulses

```
% Event 6: Firing a few pulses
%-----------------------------
k = k + 1;
event(k).type          = 'pulsed';
event(k).description   = 'Short Pulse';
event(k).aCSThruster   = [1 2];
event(k).dVThruster    = [];
event(k).dVTotal       = [];
event(k).dHTotal       = [];
event(k).disturbance   = [];
event(k).nPulses       = []; % Won't use any fuel this way
event(k).pulsewidth    = 0.05;
event(k).duration      = [];
event(k).regulated     =  0;
event(k).cM            = [0;0;0];
```

Listing 13-11 Fuel Budget Example

```
FBudget( event, x ) % Generates the following results (in 6 pt type)

TestBudget Propellant budget 19-Sep-1997

Item  Description          Total      Fuel Remaining        Fuel Used            Pressure     ACS Isp  DV Isp

  1   Initial State       1190.92     117.89    117.89     0.000     0.000    2.4e+06    2.4e+06    0.00    0.00
  2   Delta V Continuous  1153.87      99.36     99.36    18.525    18.525    2.4e+06    2.4e+06  100.02  360.00
  3   Delta V Off Pulse   1099.18      72.02     72.02    27.346    27.346    1.7e+06    1.7e+06  100.00  210.00
  4   Delta H             1099.18      72.02     72.02     0.001     0.001    1.2e+06    1.2e+06  100.41    0.00
  5   ACS Disturbance     1087.45      66.16     66.16     5.862     5.862    1.2e+06    1.2e+06  100.39    0.00
  6   Short Pulse         1087.45      66.16     66.16     0.000     0.000    1.2e+06    1.2e+06  100.50    0.00
```

For east/west stationkeeping the delta V required because of longitude drift, due to the earth's tesseral gravitational harmonics is computed by

[dVEW,dTEW] = DVLDrift(box, scLon);

where the first number is the size of the stationkeeping box in degrees and the second is the spacecraft longitude. DVLDrift outputs the delta V required per maneuver and the time between maneuvers. For north/south stationkeeping the corresponding function is

[dVNS,dTNS] = DVIDrift(box, year);

where the second argument is the year for which the calculation is done. The inclination drift is a function of the earth/moon geometry which is included in DVIDrift.

Symbols

.p 1-2

A

Acker 4-8
Ackerman's formula 4-8
Acrobat Reader 3.x 1-8
ACSDesigns 2-2
archives 1-7
ASim 9-4
AtmDens1 4-13
AtmDens2 4-5

B

beam pointing error 10-2
bisection search 10-2
Build3DSC.m 7-2, 7-10
BuildMAP.m 7-2
BuildSC.m 7-2

C

case sensitivity 1-8
CD-ROM 1-3, 2-7
cell array 5-3
CheckBoxList 16
circular error 10-2
class 5-4
  class 5-4
  constructor 5-4
  instance 5-4
  method 5-4, 5-5
  object 5-5
  overloading 5-5
  polymorphism 5-5
CLoopS 4-9
Constant 3-2
constant database 3-4
Coordinate Frame
  ECI 20
  ECIR 20
  ECR 20
CoordinateTransform 38
Cross 6-3

D

data structure 4-6, 5-2
database 3-2
DataStructureList 16
Date2JD 4-6, 4-14
delta V 10-8
DemoSC 2-2, 2-4
Disturbance.m 7-12
Disturbance.m 7-2
DrawSC.m 7-10, 7-12
DrawSC.m 7-2
DrawSCPlugin 22, 29
DSpnStab 4-7
DVIDrift 10-8
DVLDrift 10-8

E

EarthRot 4-5
EarthRotationZero 38
ECIToECIR 39
EditPM 16
EditScroll 16
ElementsPlugin 22
error checking 4-6
Eul2Q 4-12, 6-2, 6-3
Euler angle 6-3

F

FBudget 10-3
FEngine 31
FileHelp 2-5
FPlantNoise 33
FResp 4-11
FSolar 32
fuel budget 10-3

G

GASTime 39
GPMargin 4-12

Gravity Model
GEM-T1 20
JGM-2 20
JGM-3 20
WGS-84 20
GroundCoverage 40
GUIPlugIn 22
gyro 8-2

H

help 2-10
help system 2-2
HFS 1-3
hydrazine thruster 8-2

I

inclination drift 10-8
ISO 1-3

J

Jacobian 9-4
JD2000 39
JD2Date 4-14
Julian Date 39

K

Kalman Filter 8-3

L

LeadLag 4-9
List 16
ListWithButton 16
LoadCAD 7-2

M

Macintosh 1-3
MagTComp 4-7
Mat2Q 6-2
MathWorks 20
Matlab 4-6
4.x 4-5
5.x 4-5
MATLAB 1-2
Matlab Command Window 2-10
MBModel 4-12
MergeConstantDB 3-4
mex 1-2
Microwave Anisotropy Probe 8-2
monitor resolution 1-2

N

Netscape 1-2
Nichols 4-10
NORAD 9-16
north/south stationkeeping 10-8

O

ObjectAgent 1-2
online help 2-10
orbit 22
OrbitDisplayPlugin 22

P

password 1-6
PBudget 10-2
PID 8-3
PIDMIMO 8-6
PivotMch 4-15
PlanckL 4-6
Plot2D 4-14
PlotPlugIn 22
PlotV 8-13
progress bar 2-9
PropagateOrbitPlugIn 29
PSS website 1-3

Q

Q2Eul 6-3
Q2Mat 4-12, 6-3
QForm 6-2

QIToBDot 9-4
QMult 5-6, 6-2
QPose 6-2, 6-3
QTForm 6-2
quat 5-5
quaternion 6-2, 9-4

R

RBModel 9-4
rigid body 9-4
RK45 8-12
RVOrbGen 4-16

S

SC directory 1-8
SCDisturb.m 7-12
SCMat 1-8
SCTConstants.mat 3-4
ser 1-6
Series 4-10
Silicon Graphics 1-3
SkewSymm 6-3
source code 1-2
Spacecraft Control Toolbox 2-5
spacecraft longitude 10-8
spherical coordinates 9-16
star tracker 8-2
statespace 5-5
stationkeeping box 10-8
STKOrbit 39
Stuffit 1-7
SunV1 39

T

tachometer 8-3
Technical Support 1-9, 2-12
tesseral gravitational harmonics 10-8
TGimbal 2-3
Time GUI 2-4
TimePlugIn 22
TMAPSim 8-3
TPBudget 10-2
transformation matrix 6-2
transmat 5-5
tree model 9-12
TreeAdd.m 7-11
TreeBldCAD.m 7-2, 7-11
TreeCAD 7-11
TreeCAD.m 7-11
TreeSim 7-11
two-dimensional probability density function 10-2

U

uicontrol 32
UIElement 16, 17, 19
UNIX 1-3
updates 1-9
upgrades page 1-4
user id 1-6

W

Webboard 1-9, 2-12
wire model 9-6